Jan Langer

High-Level-Synthese von Operationseigenschaften

Jan Langer

High-Level-Synthese von Operationseigenschaften



Universitätsverlag Chemnitz
2011

Impressum

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Angaben sind im Internet über http://dnb.d-nb.de abrufbar.

Zugl.: Chemnitz, Techn. Univ., Diss., 2011

Technische Universität Chemnitz/Universitätsbibliothek
Universitätsverlag Chemnitz
09107 Chemnitz
http://www.bibliothek.tu-chemnitz.de/UniVerlag/

Herstellung und Auslieferung
Verlagshaus Monsenstein und Vannerdat OHG
Am Hawerkamp 31
48155 Münster
http://www.mv-verlag.de

ISBN 978-3-941003-48-4

urn:nbn:de:bsz:ch1-qucosa-79059
URL: http://nbn-resolving.de/urn:nbn:de:bsz:ch1-qucosa-79059

Bibliographische Angaben

High-Level-Synthese von Operationseigenschaften
Jan Langer
196 Seiten, 54 Abbildungen, 3 Tabellen, 4 Algorithmen, 92 Literaturstellen
Technische Universität Chemnitz, Fakultät für Elektrotechnik und Informationstechnik, Professur Schaltkreis- und Systementwurf

Schlagworte

Eigenschaftsbasierter Entwurf, Formale Verifikation, Hardwarebeschreibungssprache, Hardwaresynthese, High-Level-Synthese, Operationsbasierter Entwurf, Operationseigenschaften, Temporale Eigenschaften, Transaktionsbasierter Entwurf, Vollständigkeitsprüfung, Zeitdiagramme

Kurzreferat

In der formalen Verifikation digitaler Schaltkreise hat sich die Methodik der vollständigen Verifikation anhand spezieller Operationseigenschaften bewährt. Operationseigenschaften beschreiben das Verhalten einer Schaltung in einem festen Zeitintervall und können sequentiell miteinander verknüpft werden, um so das Gesamtverhalten zu spezifizieren. Zusätzlich beweist eine formale Vollständigkeitsprüfung, dass die Menge der Eigenschaften für jede Folge von Eingangssignalwerten die Ausgänge der zu verifizierenden Schaltung eindeutig und lückenlos determiniert.

In dieser Arbeit wird untersucht, wie aus Operationseigenschaften, deren Vollständigkeit erfolgreich bewiesen wurde, automatisiert eine Schaltungsbeschreibung abgeleitet werden kann. Gegenüber der traditionellen Entwurfsmethodik auf Register-Transfer-Ebene (RTL) bietet dieses Verfahren zwei Vorteile. Zum einen vermeidet der Vollständigkeitsbeweis viele Arten von Entwurfsfehlern, zum anderen ähnelt eine Beschreibung mit Hilfe von Operationseigenschaften den in Spezifikationen häufig genutzten Zeitdiagrammen, sodass die Entwurfsebene der Spezifikationsebene angenähert wird und Fehler durch manuelle Verfeinerungsschritte vermieden werden.

Das Entwurfswerkzeug *vhisyn* führt die High-Level-Synthese einer vollständigen Menge von Operationseigenschaften zu einer Beschreibung auf RTL durch. Die Ergebnisse zeigen, dass sowohl die verwendeten Synthesealgorithmen, als auch die erzeugten Schaltungen effizient sind und somit die Realisierung größerer Beispiele zulassen. Anhand zweier Fallstudien kann dies praktisch nachgewiesen werden.

Inhaltsverzeichnis

Abkürzungsverzeichnis

ABD Assertion-Based Design

ABV Assertion-Based Verification

AHB Advanced High-Performance Bus (Teil von AMBA)

AMBA Advanced Microcontroller Bus Architecture

ASCII American Standard Code for Information Interchange

ASIC anwendungsspezifische integrierte Schaltung - engl. Application-Specific Integrated Circuit

AST Abstrakter Syntaxbaum - engl. Abstract Syntax Tree

BDD Binary Decision Diagram

BMBF Bundesministerium für Bildung und Forschung

BMC Bounded Model Checking

BNF Backus-Naur-Form

CTL Computation Tree Logic

DEA Deterministischer endlicher Automat

DSP Digitaler Signalprozessor - engl. Digital Signal Processor

DUV Design Under Verification

FIFO First In - First Out - Speicherverfahren nach dem Prinzip einer Warteschlange

FPGA Field Programmable Gate Array

GPSV Generalisierte Plattform zur Sensordatenverarbeitung

HAAD Hierarchical Annotated Action Diagram

HDL	Hardwarebeschreibungssprache - Hardware Description Language
HLS	High-Level-Synthese - engl. High-Level Synthesis
IPC	Interval Property Checking
ITL	InterVal Language
LAD	Leaf Action Diagrams
LISP	LiSt Processing
LTL	Linear Temporal Logic
LUT	Lookup Table
MIPS	Microprocessor without Interlocked Pipeline Stages
NEA	Nichtdeterministischer endlicher Automat
OBE	Optional Branching Extension
OVA	OpenVera Assertions
OVL	Open Verification Library
PCI	Peripheral Component Interconnect
PLB	Processor Local Bus
PMA	Potenzmengenautomat
PSL	Property Specification Language
RAM	Random Access Memory
RE	Regular Expression
RTL	Register-Transfer-Ebene - engl. Register Transfer Level
SAT	Erfüllbarkeitsproblem der Aussagenlogik - engl. Satisfiability
SERE	Sequential Extended Regular Expressions
SSE	Professur für Schaltkreis- und Systementwurf

SVA SystemVerilog Assertions

SysML Systems Modeling Language

UML Unified Modeling Language

VHDL VHSIC Hardwarebeschreibungssprache

VHSIC Very-High-Speed Integrated Circuits

Vorwort

Ich möchte an dieser Stelle vor allem meinem Betreuer Prof. Ulrich Heinkel danken, der mir an der Professur für Schaltkreis- und Systementwurf (SSE) die Möglichkeit gab diese Arbeit anzufertigen und sie auf internationalen Konferenzen vorzustellen. Weiterhin möchte ich Prof. Wolfgang Kunz für die Übernahme des Zweitgutachtens danken.

Die Idee für diese Arbeit entstand im Rahmen des Projekts "Herkules", gefördert vom Bundesministerium für Bildung und Forschung (BMBF) unter dem Förderkennzeichen 01M3082. In diesem Projekt lernte ich die formale Verifikationsmethodik von OneSpin Solutions kennen, um in Zusammenarbeit mit Alcatel-Lucent Hardwarekomponenten vollständig zu verifizieren. Die ersten Gedanken zur automatisierten Synthese eines Entwurfs auf RT-Ebene aus ITL-Eigenschaften hatte ich in einem der persönlichen Gespräche mit Dr. Jörg Bormann (damals OneSpin Solutions) in Nürnberg.

Das Team am SSE hatte wesentlichen Einfluss auf das Gelingen der Arbeit. So danke ich vor allem Dr. Vasco Jerinić für die vielen fruchtbaren Diskussionen und fachlichen Anregungen in den letzten sieben Jahren. Weiterhin danke ich Thomas Horn für seine profunden grammatikalischen Kenntnisse und sein fachliches Verständnis. Ebenso danke ich Dimo Pepelyashev für das Engagement während seiner Diplomarbeit, das zur Verbesserung des Konstruktionsalgorithmus führte und Daniel Froß, dessen Arbeit zu Partikelfiltern ein wichtiger Motor in der Entwicklung von *vhisyn* war. Mein herzlicher Dank gebührt natürlich auch allen anderen ehemaligen und gegenwärtigen Kollegen und Studenten, deren Hinweise, Geduld, Ablenkungen und nicht zuletzt Freundschaft mir sehr wichtig waren. Weiterhin bedanke ich mich bei Prof. Dietmar Müller, der mir vor elf Jahren eine Stelle als studentische Hilfskraft am SSE anbot und mich seitdem in vielerlei Hinsicht unterstützt hat.

Ein besonderer Dank gilt auch den Entwicklern der wunderschönen Programmiersprache Python, die es ermöglicht hat, dass die Implementierungsarbeiten an *vhisyn* wahrscheinlich nur ein anstatt drei Jahre benötigt haben und es ein überschaubareres Stück Software geblieben ist.

Mein größter Dank gebührt meinen Eltern und meiner Schwester, die mich zu einem ehrgeizigen Menschen erzogen haben, sowie Jessica, weil sie alle immer hinter mir standen und mich unterstützt haben.

1 Einleitung und Motivation

1.1 Operationen als Entwurfsebene

Der Entwurf integrierter Schaltkreise wird in der Regel auf der Register-Transfer-Ebene (RTL) durchgeführt. Dabei kommen synchrone Taktsignale zum Einsatz, deren Flanke einen Übergang des Systems von einem Zustand zum nächsten verursacht. Die Designbeschreibung besteht dabei zum einen aus Elementen, die den Zustand speichern und Elementen, die die Übergangsfunktion realisieren. Während die speichernden Elemente (Register) bei einem Entwurf auf RTL explizit angegeben werden müssen, kann die Übergangsfunktion mit Hilfe einer Vielzahl von Konstrukten implementiert werden. Dazu zählen neben einfachen Gattern für boolesche Funktionen auch aufwendige arithmetische Operatoren, wie bspw. die Integerdivision, oder sogar Kontrollstrukturen, wie sie aus Programmiersprachen bekannt sind. Diese Operatoren und Kontrollstrukturen haben meist eine fest definierte Hardwarerepräsentation durch die sie im weiteren Verlauf des Entwurfsflusses ersetzt werden. So wird aus einer Verzweigung (**if**) auf Gatterebene ein Multiplexer und auf Schaltkreisebene eine Anzahl von Transistoren, die miteinander verschaltet sind. Die komplexeren der Kontrollstrukturen auf RT-Ebene können sogar Funktionsaufrufe und Schleifen mit einer konstanten Anzahl Durchläufe enthalten. Der Entwurf auf RTL erfordert im Regelfall nur ein begrenztes Wissen über die Technologieaspekte oder die exakte Realisierung dieser Konstrukte. Diese Trennung von Technologiewissen und Entwurf auf höheren Ebenen war ein wichtiger Schritt auf dem Weg zu der hohen Produktivität, die mit heutigen Entwurfswerkzeugen erreicht werden kann.

Ausgehend von dieser Abstraktion ist es das Ziel vergangener und aktueller Forschung, den Entwurf nicht nur vom Technologiewissen oder den Details der verwendeten Logikelemente zu entkoppeln, sondern Beschreibungselemente zu verwenden, die der ursprünglichen Intention des Entwerfers möglichst nahe kommen. Die Spezifikation ist der Ausgangspunkt des Entwurfsprozesses. Sie liegt oft in textueller Form vor. Der bislang erfolgversprechendste Ansatz, die Abstraktion des Entwurfs auf eine neue Ebene zu heben ist die Spezifikation einer Beschreibung auf algorithmischer Ebene und dessen automatisierte Übertragung auf RT-Ebene. Dieser Prozess wird im allgemeinen High-Level-Synthese (HLS) genannt. Der Algorithmus

liegt dabei in der Regel in einer etablierten Programmiersprache für Software vor, was den Vorteil bietet, dass der Entwurf einfach und schnell ausgeführt werden kann, und somit sehr kurze Entwicklungs- und Validierungszeiten entstehen. Weiterhin war es bereits ohne HLS häufig üblich, das zu erstellende Design als Softwareprototyp zu modellieren und zu evaluieren. Daher ist seitens des Entwurfsteams wenig zusätzlicher Einarbeitungs- und Migrationsaufwand notwendig.

Der große Vorteil einer Hardwareimplementierung ist die implizit vorhandene Parallelität von Hardware. Ein System versucht diese Parallelität auszunutzen, um hohe Verarbeitungsgeschwindigkeiten zu erreichen. Im Gegensatz dazu bietet eine algorithmische Beschreibung keine native Unterstützung von Parallelität an. Es muss also bei der HLS aus einem sequentiell abzuarbeitenden Algorithmus parallel arbeitende Hardware erzeugt werden. Dies wird durch zwei Ansätze bewerkstelligt. Zum einen kann das Syntheseverfahren die implizit vorhandene Parallelität extrahieren und zum anderen können vom Entwerfer bereitgestellte Annotationen im Quelltext als zusätzliche Angaben für die Parallelisierung verwendet werden. Während die HLS für datenflussorientierte Funktionalität dadurch bereits sehr gute Ergebnisse erzielt, hat man bei kontrollflussorientierter Funktionalität häufig zu wenig Kontrolle darüber, in welchem Zeitschritt welche Aktionen ausgeführt werden.

An dieser Stelle kommt die in dieser Arbeit genutzte operationsbasierte Beschreibung zum Einsatz. Jede der dabei verwendeten Operationen definiert das Verhalten des Schaltkreises während eines festen Zeitintervalls. Durch sequentielle Verkettung der Operationen wird das Gesamtverhalten modelliert. Ein Entwurf mit Hilfe von Operationen füllt eine Nische zwischen algorithmischen Beschreibungen und RTL aus, was insbesondere dann von Vorteil ist, wenn die Spezifikation das Verhalten bereits als Folge von elementaren Operationen beschreibt. In vielen Spezifikationen sind Zeitdiagramme enthalten, die genau diese Operationen darstellen. Weitere Details zu den Entwurfsebenen, der High-Level-Synthese und dem operationsbasierten Entwurf werden im nächsten Kapitel als Grundlage dieser Arbeit eingeführt.

1.2 Operationen als Verifikationsmethodik

Neben den Zeitdiagrammen ist ein weiterer wesentlicher Einfluss, der zu dieser Arbeit geführt hat, die formale Verifikation digitaler Schaltkreise mit Hilfe temporaler Eigenschaften. Bei der Firma OneSpin Solutions wurde in langer Tradition eine Methodik entwickelt, die als GapFree-Verification™ im Verifikationswerkzeug 360MV vermarktet wird und auf zwei Standbeinen beruht [OSS10]. Zum einen wird das zu verifizierende Verhalten einer Komponente

nicht, wie sonst üblich, in allgemeine temporale Eigenschaften gefasst, die oft in keinem zeitlichen Verhältnis zueinander stehen und auch in ihrer Länge nicht beschränkt sind. Stattdessen haben die Eigenschaften dieser Methodik haben einen sehr einfachen und klaren Aufbau und legen das komplette Systemverhalten in einem Zeitfenster konstanter Länge fest. Dadurch kann mittels Aneinanderreihung dieser Eigenschaften das Systemverhalten vollständig und lückenlos beschrieben werden. Die einzelnen Eigenschaften werden dann durch effektive Beweisverfahren formal geprüft.

Das zweite Standbein der Methodik ist die sogenannte Vollständigkeitsprüfung. Sie stellt fest, ob die Aneinanderreihung der Eigenschaften ein eindeutiges und deterministisches Verhalten repräsentiert. Bei einem Erfolg stellt die Menge der Eigenschaften ein Referenzmodell dar. Die formale Eigenschaftsprüfung (das erste Standbein) ist in der Lage dieses Referenzmodell mit der tatsächlichen Implementierung zu vergleichen und abweichendes Verhalten der Implementierung zu detektieren.

Das Verfahren kann damit als Ansatz verstanden werden, bei dem mit Hilfe einer speziellen Entwurfsmethodik ein goldenes Referenzmodell erstellt wird, welches, ähnlich dem Äquivalenzvergleich, mit der Implementierung verglichen wird.

Es ist also naheliegend, zu untersuchen, ob dieses Referenzmodell nicht direkt als Implementierung verwendet werden kann ohne manuelle Verfeinerungsschritte zu erfordern. Die Vorteile liegen auf der Hand. Die formal beweisbare Vollständigkeit der Eigenschaften hilft dabei Designfehler zu vermeiden. So stellt der Entwurf der Operationseigenschaften stellt in vielen Anwendungsgebieten eine höhere Abstraktion gegenüber RTL dar. Einige der Nachteile der algorithmischen Beschreibung können dabei vermieden werden. Dies betrifft vor allem die Beschreibung taktgenauen Verhaltens.

1.3 Zielstellung und Entwurfsfluss

Ziel dieser Arbeit ist die automatisierte Synthese einer effizienten Schaltungsbeschreibung auf RT-Ebene aus Operationseigenschaften zu untersuchen. Wenn das Ergebnis dieser Synthese vorliegt, können existierende Logiksynthesewerkzeuge verwendet werden, um eine Netzliste und später einen lauffähigen Hardwareprototypen zu erstellen. Die konkrete Sprache der Eigenschaften soll dabei die InterVal Language (ITL) sein, die auch bei der GapFree-Verifikation von OneSpin Solutions zum Einsatz kommt. Dadurch kann die Vollständigkeit von durchgeführten Beispielentwürfen nachgewiesen werden und die Operationseigenschaften können auf dem erzeugten Design formal geprüft werden. Damit werden das Synthesewerkzeug und die in ihm

enthaltenen Algorithmen auf ihre Zuverlässigkeit und Fehlerfreiheit hin getestet.

Im Rahmen dieser Arbeit entsteht ein experimentelles Entwurfswerkzeug *vhisyn*, das eine ITL-Beschreibung einliest und ein VHDL-Modell ausgibt. Die Hardwarebeschreibungssprache VHSIC Hardwarebeschreibungssprache (VHDL) ist neben Verilog eine der beiden mit Abstand häufigsten Sprachen, um einen Schaltkreis auf RT-Ebene zu beschreiben [IEEE09; IEEE01].

Der angestrebte Entwurfsfluss ist in Abb. 1.1 dargestellt und zeigt die Interaktion zwischen formalem Verifikationswerkzeug (OneSpin 360MV) und dem High-Level-Synthese-Werkzeug *vhisyn*. Aus der Spezifikation wird in einem manuellen Prozess eine Eigenschaftsmenge in Form einer ITL-Komponente erstellt. Diese besteht aus den eigentlichen Eigenschaften und einem Eigenschaftsgraphen, der die Interaktion der Eigenschaften erfasst. Nachdem die Komponente beschrieben wurde, muss sie auf ihre Vollständigkeit hin überprüft werden. Das Ergebnis der Prüfung ist Voraussetzung für die Synthese. Der Synthesevorgang schreibt das RT-Modell der Eigenschaftsmenge in eine VHDL-Datei. In einem letzten Schritt muss die Korrektheit des erzeugten Modells anhand der Eigenschaften formal bestätigt werden.

Die Prüfung auf Vollständigkeit kann bereits während der Erstellung der Operationseigenschaften angestoßen werden, und so dem Entwerfer frühzeitig Hinweise geben, die die Qualität des Entwurfs verbessern und Fehler aufdecken.

Weiterhin untersucht diese Arbeit die Grenzen der Synthesealgorithmen und die Effizienz der entstehenden Entwürfe im Vergleich zum Entwurf auf RT-Ebene. Dabei wird besonderes Augenmerk auf die Synthese von sehr langen Eigenschaften gelegt, da diese den Entwurf einiger Anwendungsfälle, wie bspw. rahmenorientierter Kommunikationsprotokolle, stark vereinfachen.

1.4 Aufbau der Arbeit

In Kapitel 2 werden zum einen die wichtigsten Einflussfaktoren und theoretischen Grundlagen dieser Arbeit behandelt. Zum anderen werden verwandte Arbeiten vorgestellt und mit den in dieser Arbeit vorgestellten Ansätzen verglichen.

Im darauf folgenden Kapitel 3 werden die Struktur und der Aufbau von ITL-Eigenschaften erläutert. Neben den syntaktischen Ausführungen werden auch die formalen Vollständigkeitsprüfungen kurz eingeführt. Dies bildet die Grundlage, um die in Kapitel 4 präsentierten Synthesealgorithmen verstehen zu können. Die Definition eines nichtdeterministischen und eines determinis-

tischen Kontrollautomaten werden ebenso beschrieben, wie die anschliessende Abbildung in Hardwarekonstrukte.

Es gelten verschiedene Anforderungen an eine synthesefähige ITL-Beschreibung. Diese Anforderungen werden in Kapitel 5 untersucht. Dazu zählen zum einen technische Einschränkungen, da nicht alle ITL-Konstrukte unterstützt werden, und zum anderen Anforderungen an den Beschreibungsstil, damit die Synthese von Operationseigenschaften nicht nur technisch möglich, sondern auch aus Sicht des Entwerfers sinnvoll ist.

Die Anwendung der vorgestellten High-Level-Synthese-Methodik auf zwei Anwendungsbeispiele ist Gegenstand von Kapitel 6. Weiterhin werden allgemeine Erfahrungen mit der Methodik vorgestellt. Im letzten Kapitel 7 wird die Arbeit zusammengefasst und ein Ausblick auf weitere Entwicklungsmöglichkeiten gegeben.

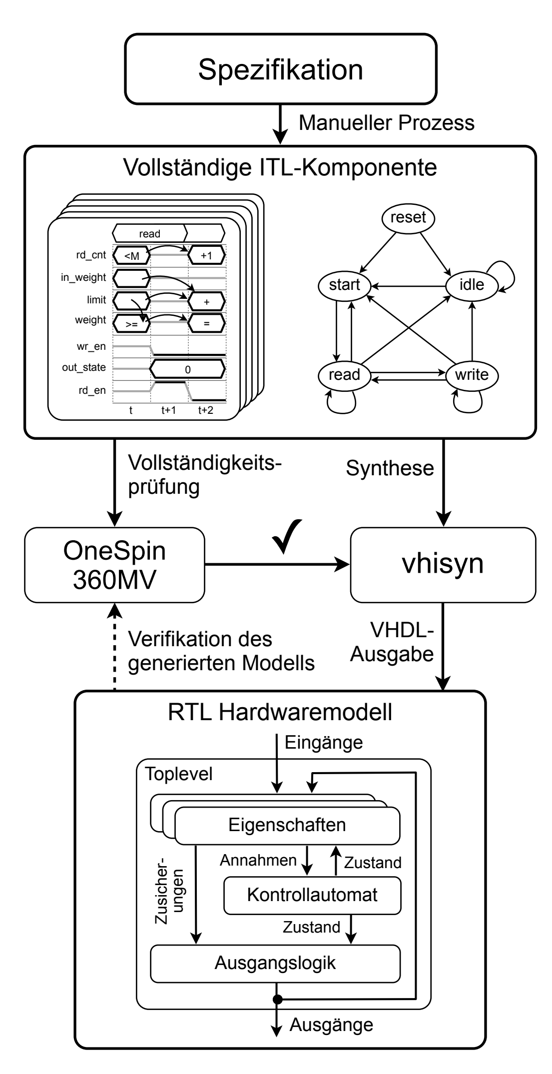


Abbildung 1.1: Entwurfsfluss bei der High-Level-Synthese von Operationseigenschaften.

2 Grundlagen temporaler Eigenschaften und ihrer Synthese

2.1 Entwurfsebenen

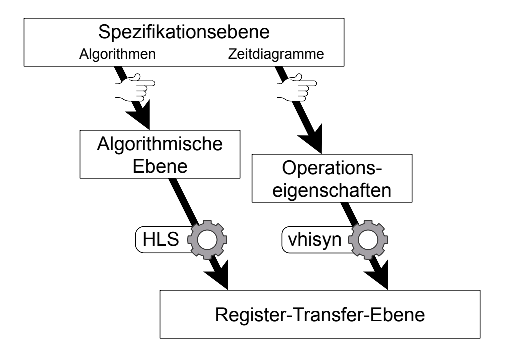


Abbildung 2.1: Einordnung des operationsbasierten Entwurfs in den Designflow.

Um die in dieser Arbeit vorgestellte Methodik zur High-Level-Synthese von Operationseigenschaften in den aktuellen Stand der Technik einordnen zu können, wird zuerst die zugehörige Entwurfsebene bestimmt. Der Entwurf digitaler Schaltungen wird oft mit Hilfe des Y-Diagramms nach [GK83] und [WT85] dargestellt. An dieser Stelle sollen nur die obersten drei Ebenen des Diagramms betrachtet werden. Diese umfassen die Spezifikations-, die algorithmische sowie die Register-Transfer-Ebene. In Abb. 2.1 sind die drei Ebenen dargestellt. Zusätzlich ist der operationsbasierte Entwurf eingefügt, mit dem sich diese Arbeit beschäftigt.

Die Spezifikation als oberste Ebene liegt meist nicht in einer formal prüf- und ausführbaren Form vor. Häufig ist sie eine vom Spezifikationsingenieur erstellte, textuelle Beschreibung und kann wiederum andere Darstellungsformen enthalten, wie z.B. Tabellen, Pseudokode oder Zeitdiagramme (auch Impulsdiagramme oder *timing diagrams*) [Hei99].

Ausgehend von der Spezifikation muss meist manuell eine Beschreibung auf der nächsten Ebene erstellt werden. So muss Pseudokode in eine konkrete Programmiersprache oder ein graphisch dargestelltes Zeitdiagramm in Operationseigenschaften überführt werden. Obwohl sich Operationseigenschaften erheblich von algorithmischen Beschreibungen unterscheiden, kann man sie in der algorithmischen Ebene ansiedeln, da sie konkreter als die Spezifikation und zugleich abstrakter als Register-Transfer-Beschreibungen sind. In der Abbildung sind Operationseigenschaften etwas niedriger dargestellt als Algorithmen, um zu verdeutlichen, dass sie einige Merkmale der RT-Ebene, wie bspw. taktgenaues Verhalten der Ein- und Ausgänge, aufweisen.

Auf algorithmischer Ebene liegt ein Entwurf meist als Programm in einer höheren Programmiersprache vor, das durch Verfahren der High-Level-Synthese (HLS) automatisiert in eine Beschreibung auf Register-Transfer-Ebene überführt werden kann [MPC88; MS09]. Das entsprechende Verfahren für Operationseigenschaften soll Gegenstand dieser Arbeit sein.

Im folgenden werden einige Zusammenhänge und wissenschaftliche Arbeiten vorgestellt, die Einfluss auf die Synthese von Operationseigenschaften haben und die Grundlage dieser Arbeit bilden.

2.2 Eigenschaftsbasierte Verifikation

Die eigenschaftsbasierte Verifikation oder auch Assertion-Based Verification (ABV) beschreibt ein Verfahren, bei dem das laut Spezifikation erwünschte Verhalten in Form von Eigenschaften beschrieben wird. In diesem Abschnitt werden Eigenschaften (engl. *properties*) gleichbedeutend mit engl. *assertions* gesehen. In einzelnen Anwendungsgebieten bestehen Unterschiede zwischen den beiden Begriffen, auf die dann an den betreffenden Stellen eingegangen wird.

Die Gültigkeit einer Eigenschaft kann anhand der konkreten Implementierung eines Designs geprüft werden. Dies geschieht auf zwei verschiedene Arten. So können zum einen klassische simulationsbasierte Verfahren angewendet werden (dynamische Verifikation). Die Eigenschaften stellen in diesem Fall sicher, dass jeder Simulationslauf die Eigenschaften erfüllt. Bei Verletzung der Eigenschaften erhält der Verifikationsingenieur eine Fehlermeldung, die zusammen mit dem Simulationstrace der Fehlersuche und -behebung dient. Die Verwendung von Eigenschaften stellt zusätzliche Informationen über die Ursache eines Fehlers bereit. Wenn bspw. das Senden eines Paketes über einen Bus fehlerhaft ist, kann mit Hilfe einer Eigenschaft, die die Bestätigung jedes Pakets prüft, der Fehler direkt entdeckt werden. Ohne Eigenschaft würde man erst zu einem späteren Zeitpunkt feststellen, dass der

Empfänger des Paketes die erwarteten Antwortdaten noch nicht geschickt hat. Wenn bei der dynamischen Verifikation eine Eigenschaft keinen Fehler meldet, kann keine Aussage getroffen werden, ob die Eigenschaft immer erfüllt ist. Es ist lediglich bekannt, dass sie in dem konkreten Testszenario (engl. *Testcase*) erfüllt ist.

Die andere Möglichkeit Eigenschaften zu prüfen ist die statische Verifikation, bzw. formale Eigenschaftsprüfung (*Model Checking*). Deren Ziel ist der mathematische Beweis, dass das Design Under Verification (DUV) in jedem Fall die Eigenschaften erfüllt. Der Beweis kann alternativ auch mit Hilfe von Theorembeweisern durchgeführt werden, wobei dann oft eine Interaktion mit dem Benutzer erforderlich ist. Letztendlich kann im Gegensatz zur dynamischen Verifikation eine definitive Aussage über die Gültigkeit der Eigenschaft getroffen werden. Nachteilig ist dabei jedoch oft das Problem der Zustandsraumexplosion, das die Anwendbarkeit der statischen Verifikation in vielen Fällen einschränkt.

Die Vorteile der eigenschaftsbasierten Verifikation sind vielfältig. Die Eigenschaften können bereits während der Spezifikationsphase erstellt werden und bilden eine zusätzliche Dokumentationsmöglichkeit, die die oft nicht eindeutigen Anforderungen der textuellen Spezifikation in formal eindeutige Konstrukte übersetzt. Eine andere Verwendungsmöglichkeit besteht während der Entwurfsphase. Die Eigenschaften müssen dabei nicht alle auf einmal hinzugefügt werden, sondern können vom Entwerfer inkrementell erstellt werden, um die soeben implementierte Funktionalität auf niedriger Ebene zu prüfen und evtl. auch zu dokumentieren. Es gibt dabei keinen Vollständigkeitsanspruch der Eigenschaften, d.h. sie decken oft nur einen Teil des Verhaltens ab, was meist als Vorteil verstanden wird, da es dem Entwerfer erlaubt, nur Verhalten in Eigenschaften zu fassen, dass durch die Semantik der Eigenschaftssprache geeignet beschrieben werden kann. Im Laufe eines Projektes kann die Anzahl der Eigenschaften auf viele hundert oder tausend anwachsen.

Ein weiterer großer Vorteil ist die Wiederverwendbarkeit von Eigenschaften. Das Verhalten von komplexen Bussystemen kann über die gesamte Projektlaufzeit und über mehrere Entwurfsebenen hinweg geprüft werden, indem ein Satz von Eigenschaften oder ein spezieller Monitor eingesetzt werden. Diese sind nur vom zu prüfenden Verhalten abhängig und können somit in verschiedenen Anwendungen eingesetzt werden. Die Eigenschaften, bzw. der sie ersetzende Monitor, erfassen dabei kontinuierlich die Werte der Bussignale und melden einen Fehler, wenn diese die Busspezifikation verletzen.

Eigenschaften können nach bestimmten Kriterien kategorisiert werden. Eine verbreitete Unterscheidung ist die zwischen Sicherheits- (*safety property*)

und Lebendigkeitseigenschaften (*liveness property*). Informell werden diese Bezeichnungen oft mit den folgenden Sätzen beschrieben:

Sicherheitseigenschaft
nothing bad ever happens
(nichts Schlimmes wird jemals passieren)

Lebendigkeitseigenschaft
something good eventually happens
(etwas Gutes wird irgendwann einmal passieren)

Anhand einer Ampelsteuerung kann eine Sicherheitseigenschaft "Hauptstrasse und Nebenstrasse dürfen nie gleichzeitig auf grün schalten" aufgestellt werden, während die Eigenschaft "Jede Strasse wird immer wieder einmal auf grün schalten" ein Lebendigkeitskriterium ausdrückt.

Sicherheitseigenschaften, die oft keine zeitlichen Zusammenhänge repräsentieren, sondern nur bestimmte unerwünschte Zustände des Systems ausschliessen sollen, werden auch Invarianten des Designs genannt und sind in vielen Hard- und Softwaresprachen bereits enthalten. So kann mit dem Schlüsselwort **assert** in VHDL eine solche Invariante in Form eines booleschen Ausdrucks definert werden. Der Simulator prüft während des Simulationslaufes die Erfüllung des Ausdrucks und meldet dem Benutzer mögliche Fehler. In der Softwarebranche sind diese Art von Eigenschaften seit vielen Jahren Stand der Technik. Die Bibliotheksfunktion **assert()** im C-Standard von 1989 oder das entsprechende Schlüsselwort in Python dienen einem ähnlichen Zweck und haben die Verifikation und das Debugging von Software deutlich vereinfacht [Hoa03]. Invarianten erfassen kein zeitliches Verhalten, sondern betrachten nur einen einzelnen Zeitschritt. So bezieht sich die Aussage "beide Ampeln sind nicht gleichzeitig grün" nur auf einen einzelnen Zeitpunkt und hat keine Auswirkungen auf die Zeitpunkte davor und danach.

Eine weitere Unterscheidung kann zwischen Eigenschaften getroffen werden, die unterspezifiziertes Verhalten aufdecken (z.B. spezielle Umgebungsbedingungen, unter denen das Design falsch reagiert) und denen, die überspezifiziertes Verhalten prüfen (z.B. Eigenschaften zu einer Funktionalität, die nie aktiviert wird). Weiterhin können Eigenschaften Verletzungen von Umgebungsbedingungen, Deadlock- oder Livelock-Situationen erkennen.

Mit dem Stichwort Assertion-Based Design (ABD) wird entsprechend [FKL04] betont, dass sich die Eigenschaften ohne Kenntnis der Implementierung direkt aus der Spezifikation herleiten und damit einen ersten und wichtigen Schritt im Design-Zyklus darstellen. Insbesondere bei der Behebung von Fehlern stellen formal exakt definierte Eigenschaften eine weitaus

bessere Spezifikation dar als Textpassagen, die oft nicht eindeutig formuliert sind.

Es stellt sich bei der eigenschaftsbasierten Verifikation die Frage, wann das Verifikationsteam genügend Eigenschaften formuliert und geprüft hat, sodass alle Aspekte des Verhaltens abgedeckt (engl. *coverage*) sind. In [CKV01] wird eine Abdeckungsmetrik definiert, die diejenigen Zustände als nicht abgedeckt betrachtet, die gelöscht werden können, ohne dass die Spezifikation in Form der Eigenschaftsmenge verletzt wird. Ein sehr weit entwickelter Ansatz, der auch die Basis dieser Arbeit bildet, ist die GapFree-Verification von OneSpin Solutions [OSS10]. Dabei werden Eigenschaften als sequentiell ablaufende Operationen betrachtet, die jeweils das Ausgangsverhalten eines Moduls während der gesamten Dauer der Operation lückenlos beschreiben. Gleichzeitig existiert ein Verfahren, um zu prüfen, ob zwischen den Operationen Lücken auftreten. Diese Tests zusammengenommen erbringen den Nachweis, dass die Eigenschaftsmenge für jede mögliche Eingangsfolge exakt eine Abfolge von Operationen bestimmt, die wiederum das Ausgangsverhalten lückenlos definieren. Damit kann es nicht mehr passieren, dass Eigenschaften "vergessen" und mehrdeutig spezifiziert werden. Es kann jedoch weiterhin falsches Verhalten in den Eigenschaften beschrieben werden.

2.2.1 Temporale Logik

Grundlage der ABV ist die temporale Logik. Sie verbindet die normale nicht-temporale Logik mit zeitbehafteten Konstrukten. Damit können Aussagen getroffen werden, die festlegen wie sich ein System über einen zeitlichen Verlauf hinweg verhält. In [Eme90] wird ein Überblick über die umfangreichen Arbeiten zu temporalen Logiken gegeben. In diesem Abschnitt soll nur auf drei spezifische Formen eingegangen werden, die sehr weit verbreitet sind und daher Eingang in die standardisierten Eigenschaftssprachen gefunden haben.

Wichtigstes Kriterium bei der Unterteilung der temporalen Logik ist die Repräsentation der Zeit. Sie kann entweder linear verlaufen oder sich verzweigen. Bei einem linearen Zeitverlauf betrachtet man genau eine Abfolge der Zustände eines Systems. Diese Abfolge wird auch als Pfad bezeichnet und entspricht einer einzelnen konkreten Ausführung des Entwurfs. Die zweite Möglichkeit ist eine verzweigende Zeit. Dabei hat das System in jedem Zustand mehrere mögliche Folgezustände, sodass ein zeitlicher Zustandsbaum entsteht, der eine unendliche Tiefe besitzt. In diesem Baum gibt es einen Wurzelknoten, der den Startzustand kennzeichnet, sowie eine unendliche Anzahl von Pfaden des Baums, die einer einzelnen linearen Ausführung des Systems entsprechen. Für eine ausführliche Behandlung der Vor- und Nachteile von

linearer und verzweigender Zeit, auf die hier nicht weiter eingegangen werden soll, kann auf [Var01] verwiesen werden.

Bei der Definition temporaler Sprachen wird nicht nach deren Verwendung zur Verifikation von Soft- oder Hardware unterschieden. Obwohl sowohl die statischen als auch die dynamischen Prüfverfahren unterschiedlich sind, werden die gleichen Sprachkonstrukte verwendet.

Reguläre Ausdrücke

Eine Form temporaler Logik leitet sich aus den regulären Ausdrücken (*regular expressions* - RE) ab. Diese wurden ursprünglich für die Mustererkennung entwickelt und sind Teil vieler Programmiersprachen, Kommandozeilentools und Texteditoren. RE sind eine Beschreibung der regulären Sprachen, die wiederum diejenigen Sprachen sind, die von den endlichen Automaten akzeptiert werden [HMU06]. Es gibt eine direkte Entsprechung und Konvertierungsmöglichkeit zwischen einem regulären Ausdruck und dem endlichen Automaten, der diese Sprache akzeptiert.

Reguläre Ausdrücke bestehen aus drei Basisoperatoren:

AB
: Die Verkettung der beiden RE A und B wird durch Aneinanderhängen der beiden Teile ohne ein Operatorsymbol gekennzeichnet. An dieser Stelle wird ein zeitlicher Bezug eingeführt, denn nachdem A vollständig erkannt wurde, wird nach B gesucht (die RE *cat* passt auf die Sequenz c, a und t).

$A|B$
: Dieser Operator stellt die Vereinigung der beiden RE A und B dar (mit $cat|cow$ wird entweder die zusammenhängende Eingangsfolge c, a und t oder die Folge c, o und w erkannt).

$A*$
: Der Kleene-Stern erkennt die RE A beliebig oft. Dabei ist auch die leere Eingangsfolge gültig (der Ausdruck $ca*t$ passt auf alle Sequenzen die mit c starten, dann keine oder beliebig viele a enthalten und mit t enden).

Zusätzlich gibt es oft eine große Menge von weiteren Operatoren, die den Komfort des Nutzers erhöhen oder dem "Abspeichern" von gefundenen Teilausdrücken dienen.

Die durch reguläre Ausdrücke beschriebenen Automaten können in der Verifikation als Monitore eingesetzt werden, die den zeitlichen Verlauf einer Simulation prüfen und einen Fehler melden, wenn die Sequenz der Signale

nicht dem beschriebenen Muster entsprechen. Die regulären Ausdrücke zählen zu den linearen temporalen Sprachen, d.h. sie nutzen keine verzweigende Zeit.

Lineare Temporale Logik

Die Definition der Linear Temporal Logic (LTL) durch [Pnu77] hat die temporale Logik erstmals für die Verifikation von Hard- und Software interessant gemacht. Sie ist durch eine fortlaufende lineare Abfolge der Zustände gekennzeichnet und ist damit besonders gut für die dynamische Verifikation geeignet, da sich in einem Simulationslauf genau ein Ausführungspfad ergibt.

Ausdrücke in dieser temporalen Logik können vier Basisoperatoren beinhalten, die im Folgenden erläutert und in Abb. 2.2 graphisch illustriert werden. Eine LTL-Formel bezieht sich immer auf einen Startzeitpunkt, der meist auf den Resetzustand des Systems gesetzt wird.

- Die nichttemporalen Operatoren $\vee$, $\wedge$ und $\neg$ verknüpfen beliebige LTL-Formeln und beziehen sich immer auf den aktuellen Zeitpunkt.
- Mit Hilfe des *globally*-Operators Gp kann geprüft werden, ob p beginnend mit dem aktuellen Zeitpunkt immer wahr ist.
- Der *eventually*-Operator Fp beschreibt, dass p entweder zum aktuellen oder irgendeinem zukünftigen Zeitpunkt wahr ist.
- Der *next*-Operator Xp drückt aus, dass der Ausdruck p zum darauf folgenden Zeitpunkt wahr ist.
- Letztendlich kann durch den *until*-Operator pUq ausgedrückt werden, dass p solange wahr bleiben soll, bis zu einem Zeitpunkt in der Zukunft q erfüllt ist.

Computation Tree Logic

LTL kann nur eine fortlaufende, nicht verzweigende Zeit darstellen. Im Gegensatz dazu gibt es verzweigende temporale Logiken [BMP81]. Mit der Computation Tree Logic (CTL) wurde in [EH82] eine verzweigende Logik vorgestellt, die jedem der bereits eingeführten LTL-Operatoren einen von zwei Pfadquantoren E oder A voransetzt. Dabei steht E (*exists*) für "es gibt einen Pfad" und A (*always*) für "entlang aller Pfade". Die Verbindung dieser Quantoren mit den Operatoren *globally* und *eventually* ist in Abb. 2.3 veranschaulicht.

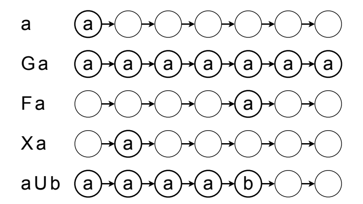


Abbildung 2.2: Graphische Darstellung von fünf einfachen LTL-Formeln, deren boolesche Teilausdrücke in den Zuständen, in denen sie gültig sind, entsprechend markiert sind [BZ08].

CTL gilt im Allgemeinen als nicht so intuitiv wie LTL. Als Illustration soll hier ein kleines Beispiel aus [Var01] dienen. Die LTL-Formel XFp bedeutet p wird irgendwann ab dem nächsten Zeitpunkt wahr. Sie ist gleichbedeutend mit der Formel FXp. Man kann diesen Ausdruck auch in CTL formulieren als $AXAFp$, allerdings kann in CTL die Formel nicht mehr umgekehrt werden, ohne dass sich die Bedeutung ändert. $AFAXp$ drückt aus, dass es in der Zukunft in jedem Fall einen Zeitpunkt gibt, zu dessen nächstem Zeitpunkt auf jeden Fall p gilt. Der Unterschied ist subtil und erfordert ein genaues Verständnis der Semantik.

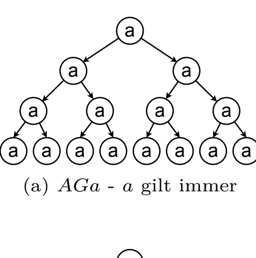
Für die statische Verifikation eignet sich CTL sehr gut, da für jede Subformel die Menge der Zustände des Systems bestimmt werden kann, in denen sie gültig ist. Damit kann die Gültigkeit einer Formel aus der Gültigkeit ihrer Subformeln hergeleitet werden. Bei LTL ist dies nicht der Fall.

Es existieren sowohl LTL-Formeln, die nicht in CTL dargestellt werden können, als auch CTL-Formeln, für die es kein LTL-Äquivalent gibt. Die Logik CTL* ist als eine Übermenge von sowohl LTL als auch CTL definiert und kann deshalb alle Formeln beider Sprachen darstellen [EH86].

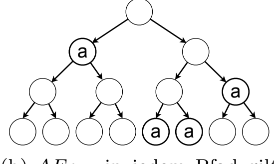
2.2.2 Eigenschaftssprachen

Die bereits beschriebenen temporalen Logiken wurden in standardisierte Sprachen übernommen. Die beiden verbreitetsten Sprachen sind die Property Specification Language (PSL) [IEEE05b] und die als Teil von SystemVerilog entstandenen SystemVerilog Assertions (SVA) [IEEE05a].

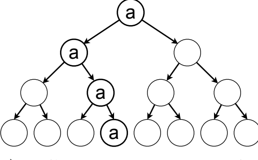
PSL entstand urprünglich bei IBM als Eigenschaftssprache Sugar. Es wurde der Standardisierungsorganisation Accellera [Acc11] für eine mögliche Standardisierung übergeben. Dabei wurden die LTL-Operatoren durch benutzerfreundlichere Schlüsselworte wie **always** und **eventually** ersetzt. PSL kann jedoch nicht nur LTL-Formeln darstellen, sondern zusätzlich auch re-



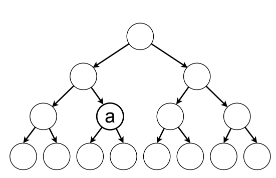
(a) AGa - a gilt immer



(b) AFa - in jedem Pfad gilt a irgendwann einmal



(c) EGa - es gibt einen Pfad, in dem a immer gilt



(d) EFa - es gibt einen Pfad, in dem a irgendwann einmal gilt

Abbildung 2.3: Vier Grundoperatoren von CTL, in deren Berechnungsbaum die Zustände, in denen a gilt, markiert sind [BZ08].

guläre Ausdrücke als Sequential Extended Regular Expressions (SERE) und CTL-Formeln in Form der Optional Branching Extension (OBE). In welchem Umfang und für welche Verwendungsmöglichkeiten einzelne Konstrukte der Sprache unterstützt werden, unterscheidet sich häufig zwischen einzelnen Anwendungen und Implementierungen.

Die zweite verbreitete temporale Sprache ist SVA. Sie kann vor allem Sequenzen beschreiben, die ähnlich regulären Ausdrücken sind. LTL-Operatoren sind für eine Aufnahme in den Standard geplant. SVA eignet sich dadurch im Moment vor allem für die dynamische Verifikation. Weitere kommerzielle Eigenschaftssprachen sind ForSpec von Intel [Arm+02], das LTL und reguläre Ausdrücke beinhaltet und als ein weiterer Kandidat bei der Standardisierung von PSL galt und *OpenVera* von Synopsys [OV11], dessen OpenVera Assertions (OVA) ebenfalls LTL-Formeln und reguläre Ausdrücke unterstützen. OpenVera bildet die Basis des SystemVerilog-Standards.

Die Hardwareverifikationssprache *e* [IEEE08] ist hauptsächlich zum Beschreiben von Testbenches gedacht, unterstützt jedoch auch auf Sequenzen basierende Assertions, die zur Laufzeit dynamisch geprüft werden. Weiterhin ist die Open Verification Library (OVL) zu erwähnen, die zwar keine temporale Sprache bereitstellt, sondern fertig implementierte Checker, die ähn-

lich temporalen Eigenschaften überprüfen, ob das Verhalten eines Entwurfs seiner Spezifikation entspricht. So existiert bspw. ein Checker der den *always*-Operator implementiert und überprüft, ob eine Bedingung immer wahr ist.

Die von der Firma OneSpin Solutions entworfene Sprache InterVal Language (ITL) entspricht im Kern einer sehr eingeschränkten Form von linearer temporaler Logik [OSS09]. Die Eigenschaften bestehen aus einer großen Implikation deren Bedingungs- und Aktionsteil jeweils aus einer Konjunktion von elementaren Prädikaten besteht, die lediglich um eine feste Anzahl Zeitschritte verschoben sein können. Aus diesem Grund umfasst die Eigenschaft nur einen begrenzten zeitlichen Rahmen (das *InTerval*). Zusätzlich bietet ITL die Möglichkeit mit Hilfe eines Eigenschaftsgraphen die Übergänge zwischen einzelnen Eigenschaften zu beschreiben. Damit kann auf einfache Art und Weise verzweigendes Verhalten dargestellt werden. In dieser Arbeit wird ITL für die Eigenschaftssynthese benutzt.

2.2.3 Dynamische Verifikation

Die dynamische Verifikation von Eigenschaften ist seit einiger Zeit Stand der Technik und ist in Simulationswerkzeugen wie Synopsys VCS, Cadence Incisive und Mentor Graphics ModelSim integriert. Die dynamische Verifikation ist im Gegensatz zu formalen Methoden nicht vom Problem der Zustandsraumexplosion betroffen und kann somit Eigenschaften auf komplexen Systemmodellen überprüfen. Die Prüfung von nur einem Ausführungspfad legt eine Spezifikation in einer linearen Logik nahe.

Bei der dynamischen Verifikation von Eigenschaften gibt es zwei grundlegende Herangehensweisen. Zum einen kann die Eigenschaft direkt mit Hilfe des Simulationstraces geprüft werden. Es kann bspw. aus dem kompletten Simulationstrace ein abstraktes Modell erstellt werden, das dann einer formalen Eigenschaftsprüfung unterzogen wird [GHS03]. Der Vorteil gegenüber einer statischen Prüfung ist die stark verringerte Komplexität des Modells. Der Test kann jedoch erst nach Beendigung der Simulation durchgeführt werden und es steht somit während des Simulationslaufes keine sofortige Information über fehlgeschlagene Eigenschaften zur Verfügung. In [Cha+03] und [Hei+03] wird die Eigenschaft zur Simulationszeit interpretiert, d.h. ein Softwaremodell der Eigenschaft läuft parallel zur Simulation und prüft mit Hilfe von Software-Events und -Threads die Gültigkeit der Eigenschaft. Da in Software eine dynamische Allokation von Berechnungs- und Speicherressourcen durchgeführt werden kann, ist dieses Verfahren sehr flexibel.

Eine andere Herangehensweise ist die Erstellung einer Hardwarerepräsentation der Eigenschaft, die als Checker oder Monitor zusammen mit dem DUV

vom Simulator ausgeführt wird. Die in der Eigenschaft verwendeten Signale sind dabei die Eingänge des Monitors, während der Ausgang ein meist binäres Signal ist, das den Erfolg oder Misserfolg der Prüfung anzeigt. Vorteilhaft ist dabei die sofortige Rückmeldung über fehlschlagende Eigenschaften während der Simulation und in einigen Fällen ein Geschwindigkeitsvorteil gegenüber der Interpretation der Eigenschaft im Simulator [Arm+06].

Auch eine Emulation des Designs zusammen mit den Eigenschaftsmonitoren bietet sich an, da die implizite Parallelität von Hardware ausgenutzt werden kann und durch die Monitore somit keinerlei zusätzliche Laufzeit verursacht wird. Die knappen Zeitressourcen bei der Produktentwicklung können dadurch weiter entlastet werden. Ebenso können die Checker auch direkt in das fertige Design integriert werden und bieten dort eine Debugging- und Überwachungsfunktionalität an. Das betrifft die gesamte Lebensdauer des Systems, angefangen beim Test nach der Fertigung bis hin zur Anwendung beim Kunden.

Von einer manuellen Implementierung des Monitors einer Eigenschaft gehen einige Nachteile aus. So kann die resultierende Beschreibung um ein Vielfaches größer und komplexer werden, als die eigentliche Eigenschaft. Dadurch wird sie fehleranfälliger und sehr schlecht wartbar. Auf Änderungen in der Spezifikation, die eine Anpassung der Eigenschaft und des Checkers erforderlich machen, kann somit nur sehr schlecht reagiert werden. Weiterhin ist die manuelle Übertragung der Operatoren einer Eigenschaft ein sehr aufwendiger Prozess, der ein hohes Expertenwissen voraussetzt.

Monitorgenerierung

Die Erzeugung eines Monitorschaltkreises für einen regulären Ausdruck ist in [HMU06] beschrieben. Aus der Regular Expression (RE) wird ein nichtdeterministischer endlicher Automat mit ϵ-Übergängen abgeleitet. Die ϵ-Übergänge sind Kanten im Automatengraph, die benutzt werden können ohne ein Eingabesymbol oder einen Zeitschritt zu verbrauchen. Sie vereinfachen den hierarchischen Aufbau des Automaten entsprechend des Syntaxbaumes der RE.

Diese Konstruktion wird auch McNaughton-Yamada-Konstruktion genannt [MY60]. Über Standardverfahren können die ϵ-Übergänge entfernt werden und der Automat in einen deterministischen, in Hardware abbildbaren Automaten überführt werden. Alternativ kann der Nichtdeterminismus auch direkt in einen Schaltkreis übertragen werden [SP01]. Die Komplexität der erzeugten Schaltung entspricht der Komplexität der RE. Diese Art der Hardwarerepräsentation eines regulären Ausdrucks findet unter anderem in der

Erkennung von schädlichen Aktivitäten in Netzwerkpaketen [MNB07] oder in der Bioinformatik beim DNA Sequenzmatching [SP01] Anwendung.

Bei der Generierung von Monitorschaltkreisen für LTL-Formeln kann entsprechend [BZ08] entweder eine automatenbasierte oder ein modulare Methodik verwendet werden. Pionierarbeit auf diesem Gebiet wurde durch das Werkzeug FoCs von IBM geleistet, das Eigenschaften in PSL bzw. dessen Vorgänger Sugar verarbeiten kann [Aba+00]. Es gilt als Benchmark für viele weitere Monitorgeneratoren. Aufgrund seiner kommerziellen Natur ist wenig über seine innere Funktionsweise bekannt. Ursprünglich wurden Monitore als Software erzeugt, die im Simulator die Eigenschaft überprüfen. In späteren Versionen ist allerdings auch eine Ausgabe der Automaten als VHDL- oder Verilog-Beschreibungen vorgesehen. Damit sind auch Anwendungen in der Emulation bzw. im Feldeinsatz möglich. FoCs unterstützt nicht den kompletten Funktionsumfang von PSL. So existiert keine Konvertierung für OBE, der bereits genannten Erweiterung von PSL für CTL-Formeln.

Eine weitere Arbeit zur automatenbasierten Monitorgenerierung ist das Werkzeug MBAC von Boulé und Zilic [BZ08]. Dieser Ansatz verwendet Ersetzungsregeln um die umfangreiche Syntax von PSL auf wenige grundlegende Algorithmen abzubilden. Das gesamte sogenannte *simple subset* von PSL wird unterstützt.

Unter den modularen Verfahren ist vor allem der Monitorgenerator Horus der Gruppe um Morin-Allory und Borrione zu nennen [MB06]. Deren Algorithmus generiert für jeden Operator einer Untermenge von PSL einen in einer Bibliothek gespeicherten Basisblock. Jeder Block hat Aktivierungseingänge, die den Start der Prüfung für diesen Operator anzeigen. An den Ausgängen des Blocks ist das Ergebnis der Prüfung abzulesen. Durch eine geschickte Verschaltung der Basisblöcke entsprechend der Syntax einer Eigenschaft kann das gewünschte Gesamtverhalten erreicht werden.

2.2.4 Statische Verifikation - Formale Eigenschaftsprüfung

Die statische Verifikation von Eigenschaften, das Model Checking [CGP99], verlangt keine Ausführung des Modells mit Hilfe eines Simulators. Stattdessen wird eine formale Analyse durchgeführt, die im Ergebnis anzeigt, ob die Eigenschaft für jede mögliche Ausführung des Modells gilt. Aus diesem Grund ist es nicht notwendig spezielle Testszenarien zu erstellen, die jeweils nur einen kleinen Teil der Funktionalität überprüfen. Durch den mathematischen Beweis der Eigenschaft werden alle Grenzfälle (engl. *Corner Cases*) automatisch berücksichtigt. Allerdings ist die formale Verifikation meist sehr aufwendig und es gelten enge Beschränkungen bezüglich der Komplexität des zu verifizierenden Systems. Die ersten Versuche des Model Checking bestan-

den aus manuell durchgeführten Beweisen von Softwareprogrammen (siehe [Eme08]).

Das Problem der Zustandsraumexplosion ist der maßgebliche Faktor, der eine schnelle Verbreitung der formalen Eigenschaftsprüfung bisher behindert hat. So verdoppelt jedes zusätzliche Register in einer zu prüfenden Schaltung die Anzahl der Zustände, und somit den Verifikationsaufwand.

Symbolic Model Checking

Der große Durchbruch bestand im symbolischen Model Checking von CTL-Formeln, bei dem der Zustandsraum symbolisch repräsentiert wird, bspw. in Form von binären Entscheidungsbäumen [Bur+90; Bry86]. Damit war es erstmals möglich Schaltungen zu verifizieren, die über kleine Anschauungsbeispiele hinausgingen. Trotz vieler Bemühungen auf diesem Gebiet kann diese Art der formalen Verifikation weiterhin nur mit begrenzten Schaltungsgrößen umgehen.

Bounded Model Checking

Bounded Model Checking (BMC) kann größere Schaltungskomplexitäten verarbeiten [Bie+99]. Dazu wird das Design über k Schritte beginnend mit dem Resetzustand sequentiell abgerollt. Im Anschluss wird die Eigenschaft über diesem eingeschränkten Zeitfenster verifiziert. Wenn die Prüfung fehlschlägt, erhält man ein Gegenbeispiel. Im anderen Fall jedoch kann man nur die Aussage treffen, dass innerhalb von k Zeitpunkten nach Reset die Eigenschaft erfüllt ist. Man erhöht k in einer weiteren Iteration und versucht wieder ein Gegenbeispiel zu erzeugen. Eine Erhöhung von k wird solange durchgeführt, bis der Prüfaufwand die zur Verfügung stehenden Ressourcen übersteigt. BMC kann eine erfüllte Eigenschaft nicht beweisen, sondern nur inkorrekte Eigenschaften widerlegen.

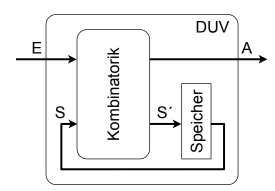
Die Funktionsweise von BMC beruht auf einer Umsetzung der abgerollten Schaltung und der Eigenschaft in ein Erfüllbarkeitsproblem (SAT). Dieses ist zwar NP-vollständig, aber es existieren effiziente heuristische Algorithmen dafür [DP60; DLL62].

Interval Property Checking

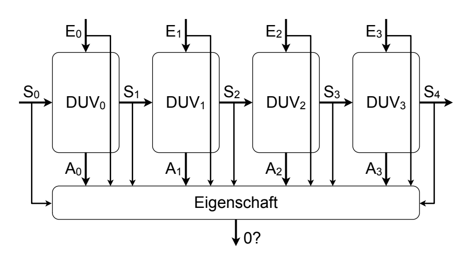
Die Firma OneSpin Solutions vertreibt eine Interval Property Checking (IPC) genannte Methodik zur Eigenschaftsprüfung, die ähnlich BMC ein Design über eine endliche Anzahl Zeitschritte abrollt und zusammen mit der Eigenschaft als SAT-Problem darstellt [OSS10]. Es gibt zwei wesentliche Unterschiede gegenüber BMC. Zum einen geht IPC nicht vom Resetzustand aus,

sondern von einem beliebigen Zustand. Zum anderen sind die Eigenschaften in ITL geschrieben, sodass sie sich nur auf ein beschränktes Zeitfenster auswirken. Die Schaltung wird dann exakt auf dieses Zeitfenster abgerollt. Die inkrementelle Erhöhung der Länge des Zeitfensters ist dabei nicht notwendig.

Falls die Prüfung des Erfüllbarkeitsproblems erfolgreich durchgeführt wurde, ist die Eigenschaft im Gegensatz zu BMC unter allen Umständen erfüllt. Bei einer gescheiterten Prüfung kann entweder ein Fehler im Design vorliegen, oder das erzeugte Gegenbeispiel geht nicht von einem erreichbaren Systemzustand aus. Abhilfe schafft hier die zusätzliche Definition von Invarianten (in ITL *assertions* genannt), die der Prüfung Erreichbarkeitsinformationen zur Verfügung stellen. Die Automatisierung dieses Prozesses ist Gegenstand aktueller Forschung [Ngu+08].



(a) Schematische Darstellung eines synchronen Entwurfs



(b) Über vier Zeitschritte abgerolltes Design mit verknüpfter Eigenschaft

Abbildung 2.4: Interval Property Checking.

In Abb. 2.4 ist dieses Verfahren graphisch dargestellt. Ein synchrones Design enthält einen kombinatorischen Block und Speicherelemente. Die Eingänge E und der gegenwärtige Systemzustand S werden vom kombinatorischen Block verarbeitet. An den Ausgängen A und dem Folgezustand S' stehen noch im selben Zeitschritt neue Werte bereit. Die Speicherelemente verzögern und speichern den Folgezustand, sodass er im nächsten Zeit-

punkt den gegenwärtigen Zustand S darstellt. Für das IPC wird nun der kombinatorische Block eines Designs ohne Speicher herausgetrennt und in der entsprechenden Anzahl vervielfältigt. Der Folgezustand einer Instanz des Designs wird mit dem gegenwärtigen Zustand der darauf folgenden Instanz verknüpft. Die Eingänge aller Instanzen und der initiale Systemzustand S_0 sind freie Signale, während die anderen Systemzustände und die Ausgänge von den Kombinatorikblöcken getrieben werden. Die Eigenschaft selbst wird auch in Hardware umgesetzt und kann alle Eingänge, Ausgänge und internen Signale benutzen um einen Wahrheitswert zu erzeugen, der bei erfüllter Eigenschaft 1 ergibt und 0, falls die Eigenschaft fehlschlägt. Im IPC wird nun dieser Ausgang der Eigenschaft fest auf 0 gesetzt und das Gesamtproblem an einen Erfüllbarkeitslöser übergeben. Wenn dieser eine Lösung findet, stellen die Werte eine gültige Belegung aller Signale dar, die die Eigenschaft verletzt, das Gegenbeispiel.

2.3 High-Level-Synthese

2.3.1 Klassische High-Level-Synthese

Im Allgemeinen wird unter High-Level-Synthese die Übertragung von algorithmischen Beschreibungen in Hardware verstanden. Die dabei verwendeten Eingangssprachen sind meist imperative Programmiersprachen, die schon im Bereich der Softwareentwicklung etabliert sind. Die verbreitetsten Vertreter sind die Sprachen C bzw. C++. Für sie gibt es bereits einige ausgereifte Ansätze für die Erzeugung von Hardware. Dazu zählen unter anderem die Software Catapult C von Mentor Graphics oder Synphony C Compiler von Synopsys. Nicht nur C/C++ dient als Ausgangspunkt eines HLS-Verfahrens. Auch aus der Sprache Matlab der Firma Mathworks im Zusammenspiel mit dem Simulations- und Modellierungstool Simulink und dem Werkzeug Stateflow für die Modellierung von Kontrollautomaten kann Hardware erzeugt werden. Zusätzlich bieten diese Werkzeuge die Möglichkeit alternativ C-Kode für DSPs zu erzeugen.

Die Aufgabe der HLS besteht in der Abbildung eines sequentiell abzuarbeitenden Softwareprogramms auf parallel arbeitende Hardware. Dabei wird das Programm in einen Kontroll- und einen Datenfluss unterteilt. Der Kontrollfluss wird vor allem durch die Kontrollkonstrukte der Programmiersprache, wie Schleifen, Verzweigungen und Auswahlanweisungen bestimmt, während der Datenfluss aus den Ausdrücken und Zuweisungen hervorgeht. Die HLS hat im Wesentlichen zwei Probeme zu lösen. Zum einen muss festgelegt werden, welche Funktionseinheiten benötigt werden, bzw. wie die abstrakten Operatoren der Programmiersprache auf diese Funktionseinheiten abgebildet

werden. Dieser Schritt wird als Allokation bezeichnet. Die andere Aufgabe ist die Planung, in welchem Zeitschritt eine Operation in der zugehörigen Funktionseinheit ausgeführt wird. Dieser Schritt wird Scheduling genannt. Die Zeitschritte korrespondieren dabei mit den Zuständen des Automaten, der den Kontrollfluss abbildet. Ziel beider Teilaufgaben der HLS ist die Minimierung sowohl der Anzahl der benötigten Funktionseinheiten (Fläche des Schaltkreises) als auch der benötigten Zeitschritte um das Programm abzuarbeiten (Durchsatz). Es kann dabei entweder die verfügbare Anzahl Funktionseinheiten oder die maximal nutzbare Anzahl Zeitschritte begrenzt werden, je nachdem ob der Fokus der Synthese auf einem hohen Durchsatz oder einem geringen Ressourcenverbrauch liegt.

2.3.2 High-Level-Synthese aus temporalen Eigenschaften

In Abschnitt 2.2 wurden Spezifikationen basierend auf temporaler Logik für die Verifikation eines Schaltkreises verwendet. Es ist jedoch in einigen Fällen wünschenswert aus einer solchen Spezifikation auch automatisiert ein Design zu erstellen. Dieses erfüllt per Definition die Eigenschaften und kann eventuelle Freiheitsgrade, die von den Eigenschaften nicht abgedeckt werden, zur Optimierung heranziehen. Die Eigenschaften werden auf einer abstrakteren Ebene und mit größerem Bezug zur textuellen Spezifikation geschrieben als eine mögliche RTL-Implementierung. Daher verringert sich die Wahrscheinlichkeit von Fehlern und die Wartbarkeit des Designs wird erhöht. Die HLS aus temporalen Eigenschaften darf nicht mit der Monitorgenerierung verwechselt werden, bei der keine Werte für Ausgangssignale generiert werden, sondern nur ein einzelnes Signal, das anzeigt, ob die Eigenschaft fehlgeschlagen ist.

Die Synthese von digitalen Designs aus einer formalen Spezifikation wird als *Church's problem* [Chu63] bezeichnet und mit [BL69] und [Rab72] wurden zwei Lösungsansätze vorgeschlagen. In [PR89] wurde ein Verfahren präsentiert, welches aus LTL-Formeln nichtdeterministische Büchi-Automaten erstellt und diese mit Hilfe des Determinierungsverfahrens von [Saf88] in deterministische Rabin-Automaten überführt. Die dabei auftretende doppelt exponentielle Laufzeit schränkt die Anwendbarkeit jedoch stark ein, da keine Beispiel-Spezifikationen implementiert werden können, die über ein paar wenige Zustände hinausgehen.

Durch Beschränkung der unterstützten LTL-Formeln auf Formeln mit dem Aufbau

$$(GFp_1 \wedge GFp_2 \wedge ... \wedge GFp_m) \rightarrow (GFq_1 \wedge GFq_2 \wedge ... \wedge GFq_n)$$

und alle Formeln, die in diese Form überführt werden können, kann ein Algorithmus angegeben werden, der in kubischer Zeit ein passendes Design synthetisiert [PP06]. Obwohl diese Einschränkung sehr restriktiv erscheint, können viele praktische Spezifikationen umgesetzt werden und in [Blo+07] ist es gelungen einen Arbiter für das AMBA AHB Busprotokoll aus einer PSL-Spezifikation zu erstellen. Trotz dieser Erfolge ist die Laufzeit dieser Algorithmen für viele praktische Anwendungen noch zu groß.

Basierend auf den Ideen der modularen Monitorgenerierung des Horus-Werkzeugs wurde in [OMB09] ein Verfahren entwickelt, das in linearer Zeit aus dem *simple subset* von PSL einen Schaltkreis synthetisiert. Es ähnelt dem in dieser Arbeit beschriebenen Verfahren, indem es jede Eigenschaft einzeln in einen Hardwareblock übersetzt. Diese Blöcke legen fest, ob die Eigenschaft im aktuellen Zeitpunkt ein Ausgangssignal schreibt und welchen Wert sie schreibt. Ein Ausgangsmultiplexer auf oberster Ebene wählt dann aus, welcher Wert dann tatsächlich an das Ausgangssignal weitergeleitet wird. Der innere Aufbau eines Eigenschaftsblockes orientiert sich wie in Horus am Syntaxbaum der Eigenschaft und verknüpft Basisblöcke aller genutzten PSL-Operatoren.

CanDo

Einen völlig anderen Weg wählt die Arbeit von [Sch09]. Darin wird die temporale Sprache, die synthetisiert werden kann, auf eine globale Implikation der Form $G(A \to P)$ beschränkt, wobei A und P Ausdrücke sind, die aus logischen Verknüpfungen der Signale des Systems zu unterschiedlichen Zeitpunkten bestehen. Diese Zeitpunkte sind relativ zu einem Zeitpunkt t definiert, der einen beliebigen Zeitpunkt der Systemausführung symbolisiert. Diese Einschränkung kann in der Praxis den Eigenschaften in ITL gleichgesetzt werden, deren Synthese auch das Ziel dieser Arbeit ist.

Die Eigenschaften werden einem Normalisierungsverfahren unterworfen, das die Eigenschaften aufteilt, sodass der Zusicherungsteil P nur einen Ausdruck enthält, der einem Signal zum Zeitpunkt $t + 1$ einen Wert zuweist. Dabei wird weitestgehend auf Wortebene gearbeitet. Die Signale, die im Annahmenteil A verwendet werden, dürfen nur zu den Zeitpunkten kleiner oder gleich $t + 1$ benutzt werden. Die normalisierte Eigenschaftsmenge wird auf eine Hardwarerepräsentation abgebildet, die aus Multiplexern besteht, deren Schaltsignal dem vorderen Teil der Implikation entspricht, während einer der Multiplexereingänge die Zusicherung darstellt. Das zugewiesene Signal ist der Ausgang des Multiplexers und wird über ein Register geleitet.

Die dabei entstehenden Schaltkreise werden Cando-Objekte genannt und beinhalten im Falle eines unvollständigen Eigenschaftssatzes Freiheitsgrade,

die über freie Signale realisiert sind, deren Wert zufällig bestimmt wird. Damit implementieren Cando-Objekte alle möglichen Realisierungen des Eigenschaftssatzes. Falls dieser jedoch vollständig ist und für jedes Ausgangssignal in jeder Situation einen eindeutigen Wert bereitstellt, wird auch das generierte Modell ein eindeutiges Verhalten aufweisen.

Die Laufzeit des Verfahrens ist mit Hilfe der vorhandenen Literatur schwer abzuschätzen. Der Algorithmus verlangt, dass sich die Annahmen von Eigenschaften, die das selbe Signal zuweisen, gegenseitig ausschliessen. Wenn das Verfahren auf Bitebene durchgeführt wird, ist diese Information meist einfach zu berechnen. Allerdings ist die Umwandlung von komplexen Datenpfadoperationen in die Bitebene sehr aufwendig. Falls auf Wortebene gearbeitet wird, werden in [Sch09] BDD-Repräsentationen und ähnliche Verfahren genannt, die in vielen Fällen lange Laufzeiten vermuten lassen.

Es werden hauptsächlich kleinere Beispiele synthetisiert, wie bspw. Master- und Slave-Komponenten eines AMBA AHB Businterfaces und ein Cache Controller für einen MIPS Core Prozessor. Die Synthese des Cache Controllers benötigt zirka 30 Minuten, wobei die Normalisierung den größten Teil der Zeit einnimmt. In der vorliegenden Arbeit wird argumentiert, dass die Verwendung des in ITL bereitgestellten Eigenschaftsgraphen und der damit einhergehende Verzicht auf jegliche boolesche Prüfverfahren in Bezug auf die Teilausdrücke der Eigenschaften zu erheblichen Geschwindigkeitsvorteilen bei Eigenschaftsmengen mit komplexen Datenpfadoperationen führt.

Ein weiterer Nachteil, der aus der Vernachlässigung des Eigenschaftsgraphen bei der Synthese von Cando-Objekten resultiert, ist die Notwendigkeit konzeptionelle Zustände in den Eigenschaften zu definieren. Diese tauchen meist in den Zeitdiagrammen, die die Grundlage für ITL-Eigenschaften bilden, nicht auf und können implizit aus dem Eigenschaftsgraph hergeleitet werden.

2.3.3 Funktionale Programmierung und die Synthese von Hardware

Im Gegensatz zu der bekannten imperativen Programmierung in bspw. C oder C++ unterliegen funktionale Sprachen einem völlig anderen Paradigma. Grundlage ist das λ-Kalkül von [Chu32], aus dem später industriell eingesetzte Programmiersprachen wie Haskell, LISP und Erlang entstanden sind. Das besondere an funktionalen Sprachen ist ihre völlige Freiheit von Seiteneffekten innerhalb von Funktionen. Wenn eine Variable einmal einen Wert zugewiesen bekommen hat, ändert sich dieser während der Programmlaufzeit nicht mehr. Diese Seiteneffektfreiheit hat verschiedene Vorteile. So sind funktionale Programme besser verifizierbar, da ihre Definition einer Funktion mehr der mathematischen Sichtweise entspricht, als die der imperativen

Programmierung. Der Aufruf einer Funktion ergibt bei gleichen Argumenten immer das selbe Ergebnis. Damit lassen sich bspw. mathematische Beweise der Korrektheit eines Algorithmus führen. Weiterhin ist die sogenannte *lazy evaluation* möglich, die die Auswertung eines Funktionsargumentes erst dann veranlasst, wenn es in einem anderen Ausdruck benötigt wird.

Bei der Synthese eines funktionalen Programms in synchrone Hardware können die benutzten Variablen direkt in Signale und die eingesetzten Operatoren in Hardware-Funktionseinheiten übersetzt werden. In [She05] wird ein Überblick über den Stand der Technik bei der Beschreibung und Verifikation von Hardwareschaltkreisen mit Hilfe funktionaler Programmierung gegeben. Erwähnenswert ist die Sprache μFP, die in [She84] für die Synthese von Hardware eingesetzt wurde. Sie basiert auf Arbeiten von [Bac78], in denen funktionale Sprachen als eine Lösung für viele Probleme der imperativen Programmierung vorgeschlagen wurden. Das Werkzeug Lava ist eine Implementierung von μFP in Haskell [Cla01]. In der gleichnamigen Hardwarebeschreibungssprache Lava wird das Verhalten eines Schaltkreises in einer Funktion beschrieben. Die Funktion selbst generiert jedoch keine Hardware. Erst ihre Benutzung durch die Funktion **writeVhdl** erzeugt die eigentliche Hardware. Ebenso existiert eine Funktion **simulate**, die eine Ausführung des Designs mit zusätzlich übergebenen Testmustern erlaubt. Durch diesen Schritt ist die Verhaltensbeschreibung mehr ein Generator der Hardware als die Hardware selbst.

An einem Beispiel aus [CS00] wird dies deutlich. Die Lava-Funktion in Abb. 2.5 implementiert einen Ripple-Carry-Addierer mit Hilfe eines Aufrufes von **halfAdd** für das erste Bit **a** und dem rekursiven Aufruf von **bitAdder** für die restlichen Bits in **as**. Diese Beschreibung ist zum einen völlig generisch bezüglich der verwendeten Datentypen, solange sie vom Halbaddierer weiterverarbeitet werden können, und sie enthält keine Schleifen oder sonstige Kontrollanweisungen mit temporären Zählervariablen und ähnlichen Konstrukten. In Abb. 2.6 ist die entsprechende Hardwarerepräsentation dargestellt.

Lava ist eine eingebettete domänenspezifische Sprache (engl. *domain specific embedded language*) und kann deshalb nicht den vollen Umfang der Wirtssprache benutzen. Ein neuerer Ansatz, der ebenfalls auf Haskell basiert, nennt sich CλaSH und drückt Hardwarekonstrukte direkt mit nativem Haskell aus [Kup+10]. Der Standard-Haskellcompiler erzeugt aus der Beschreibung eine Repräsentation in einem Zwischenformat, die eine wesentlich geringere Komplexität hat als die Ausgangssprache. CλaSH führt anschliessend eine Transformation des Zwischenformats in Hardware durch.

```
bitAdder (carryIn, []) = ([], carryIn)

bitAdder (carryIn, a:as) = (sum:sums, carryOut)
  where
    (sum, carry) = halfAdd (carryIn, a)
    (sums, carryOut) = bitAdder (carry, as)
```

Abbildung 2.5: Lava-Quelltext einer rekursiven Funktion **bitAdder**, die zwei Operanden bitweise addiert.

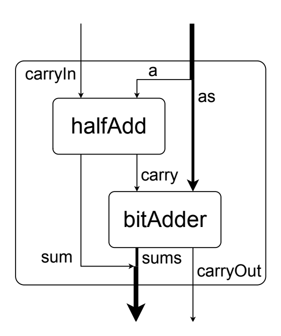


Abbildung 2.6: Hardwarebeschreibung der **bitAdder** Funktion.

Weitere Ansätze der Hardwareverifikation und -synthese mit funktionalen Sprachen sind unter anderem in [BH97] und [CLM98] beschrieben.

Der Bezug von funktionaler Darstellung von Hardwarebeschreibungen zur Eigenschaftssynthese aus ITL besteht in der von ITL verwendeten Syntax für Makros. Sie nutzen funktionale Sprachkonstrukte, bieten jedoch nicht die Möglichkeit Funktionen als Funktionsargumente zu verwenden. Makros sind einer der Gründe, warum sich ITL nicht nur gut für Kontrollflussbeschreibungen eignet, sondern auch für komplexe Datenflussoperationen, die dadurch meist sehr elegant und gut wartbar sind. Die Synthese dieser Makros orientiert sich an der angeführten Literatur.

2.4 Operationen als Verhaltensbeschreibung

Ein weiterer wichtiger Hintergrundgedanke bei der Synthese von Operationseigenschaften ist die Unterteilung von Verhalten in sequentiell ablaufende

Operationen, die in einem festgelegten Zeitfenster das Systemverhalten bestimmen. Für verschiedene Anwendungsszenarien wie bspw. die Beschreibung von Kommunikationsprotokollen ist diese Sicht auf Verhalten sehr vorteilhaft. So werden in textuellen Spezifikationen häufig Zeitdiagramme eingebunden. In Abb. 2.7 ist das Zeitdiagramm der einfachen Leseoperation aus der PCI-Spezifikation dargestellt. Die Aufteilung zwischen Annahmen über das Umgebungsverhalten und Signalzuweisungen, den Zusicherungen, ist nur implizit gegeben, da der Nutzer weiss, welche Signale gelesen und welche geschrieben werden. Wenn Zeitdiagramme in Spezifikationen verwendet werden, sind sie meist zusätzlich im Text näher erläutert.

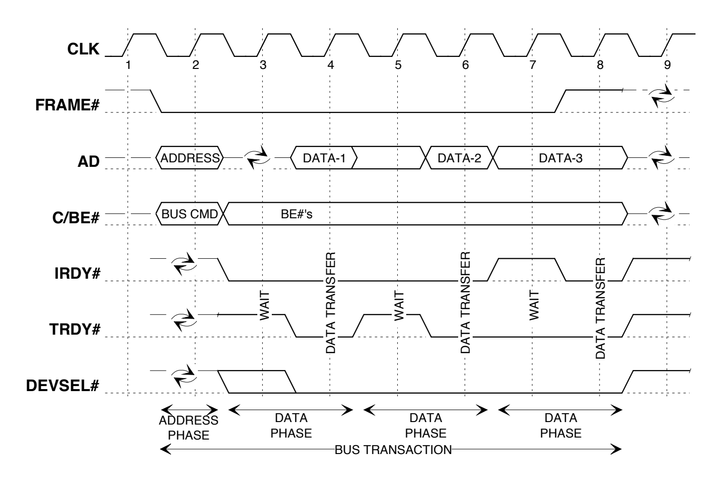


Abbildung 2.7: Zeitdiagramm der Leseoperation der PCI-Protokollspezifikation [PCI95].

In [Hei99] werden Zeitdiagramme formal erfasst und validiert. Dabei werden die Bedingungen eines Zeitdiagramms, bspw. welcher Signalwechsel einen anderen bedingt und wieviel Zeit zwischen beiden vergehen darf, definiert. Mit Hilfe dieser Angaben ist es möglich die Konsistenz eines Zeitdiagramms zu prüfen. Wenn sich die Bedingungen und dazugehörigen Intervalle gegenseitig widersprechen, liegt ein Spezifikationsfehler vor.

2.4.1 Unified Modeling Language

Es gibt Ansätze die Spezifikation von Zeitdiagrammen zu formalisieren und ihnen eine exakt definierte Semantik zu geben. Der wahrscheinlich bekann-

teste ist die Unified Modeling Language (UML). Sie ist eine standardisierte graphische Sprache, die verschiedene Diagrammtypen für die Modellierung von Softwaresystemen bereitstellt. Es wird zwischen Strukturdiagrammen und Verhaltensdiagrammen unterschieden. Zu den Verhaltensdiagrammen zählen unter anderem die Sequenz-, Zeitverlaufs- und Interaktionsübersichtdiagramme. Die Zeitverlaufsdiagramme entsprechen dabei im Wesentlichen den erwähnten Zeitdiagrammen, während Sequenzdiagramme sich auf den Austausch von Nachrichten zwischen verschiedenen Partnern konzentrieren. Die Interaktionsübersichtsdiagramme stellen eine Möglichkeit dar, mehrere Sequenz- oder Zeitverlaufsdiagramme zu kombinieren und hierarchisch zu ordnen. So gibt es verschiedene Verknüpfungoperatoren, wie bspw. Verkettungen, Verzweigungen oder parallele Verläufe. Mit Hilfe der Interaktionsübersichtsdiagramme kann mit einzelnen Unterdiagrammen (Operationen) ein Gesamtverhalten modelliert werden. Grob betrachtet entsprechen sie den Eigenschaftsgraphen von ITL-Operationen.

Die Diagrammtypen in UML sind nicht in erster Linie dazu gedacht aus ihnen die Implementierung des Systems automatisiert im Sinne eines Synthesevorgangs zu generieren. Weiterhin ist UML vor allem in der Softwareindustrie verbreitet und an deren spezielle Bedürfnisse angepasst. Eine ebenfalls standardisierte, von UML abgeleitete Modellierungssprache für den Systementwurf ist die Systems Modeling Language (SysML) [OMG10]. Diese hat jedoch sowohl Zeitverlaufsdiagramme als auch Interaktionsübersichtsdiagramme nicht aus UML übernommen, weil Bedenken bezüglich der Eignung für den Systementwurf bestanden.

2.4.2 Hierarchical Annotated Action Diagram

Ein ähnlicher Ansatz ist in [Cer+98] beschrieben. Es wird eine formalisierte Form der Zeitdiagramme definiert, die sogenannten Leaf Action Diagrams (LAD), die jeweils aus einer Menge von Aktionen und zeitlichen Beziehungen zwischen diesen Aktionen bestehen. Eine Aktion weist einem Signal einen Wert zu. Wenn das Signal ein Eingangssignal ist, entspricht die Zuweisung dem Verhalten der Umgebung, während eine Ausgangssignalzuweisung das System selbst beschreibt. Die Beziehungen zwischen den Aktionen stellen jeweils einen kausalen Zusammenhang dar, d.h. eine Aktion kann eine andere auslösen. Zusätzlich muss ein Intervall angegeben werden, in dem die zweite Aktion auf die erste folgt. Jede dieser Beziehungen kann entweder als *assume*, *commit* oder *requirement* klassifiziert werden. Durch diese formale Notation kann das Verhalten eines Zeitdiagramms mathematisch erfasst werden. Weiterhin können mehrere LAD zu einem Hierarchical Annotated

Action Diagram (HAAD) zusammengefasst werden. Dieses verknüpft einzelne LAD durch vier mögliche Operatoren.

1. Der Verkettungsoperator $A \bullet B$ drückt aus, dass LAD B unmittelbar nach dem Ende von LAD A gestartet wird.

2. Schleifen können mit Hilfe des Operator $@A$ definiert werden, der A solange hintereinander ausführt, bis eine *exit*-Bedingung erfüllt ist.

3. Der Operator für Nebenläufigkeit wird durch $A \parallel B$ beschrieben und sagt aus, dass A und B beide parallel ausgeführt werden.

4. Letztendlich bietet der Auswahloperator $A + B$ eine Verzweigung, bei der zwar beide LAD parallel gestartet werden, jedoch nur eines erfolgreich beendet werden darf.

Im Wesentlichen entspricht diese Beschreibung den Möglichkeiten, die der Eigenschaftsgraph bietet. In den verfügbaren Veröffentlichungen wird nicht auf eine formale Vollständigkeitsprüfung eingegangen. Ansonsten hat dieser Ansatz viele Gemeinsamkeiten mit den in dieser Arbeit beschriebenen Operationseigenschaften.

Obwohl keine Synthese von HAAD bekannt ist, wird in [OC99] ein Monitor generiert, der überprüft, ob das Verhalten eines Systems dem HAAD entspricht. Im Falle einer Nichtübereinstimmung wird dies als Fehler signalisiert.

2.4.3 Operationen im Entwurf

VHDL-Prozesse mit mehreren wait-Anweisungen

Ein schönes Beispiel dafür, dass das Denken in Operationen bereits im Entwurf von Schaltkreisen Anwendung findet, ist der aktuelle Standard für synthesefähiges VHDL [IEEE04]. Er erlaubt in einem synchronen sequentiellen Prozess die Verwendung von mehreren **wait**-Anweisungen, die auf dieselbe Taktflanke reagieren. Mit Hilfe dieses Konstruktes kann in einem Prozess ein Verhalten beschrieben werden, das sich über mehrere Taktzyklen erstreckt und damit sozusagen eine Operation im Sinne dieser Arbeit darstellt. Das Synthesewerkzeug ist bei einer solchen Beschreibung gezwungen, einen impliziten Automaten zu erstelllen, der den Kontrollfluss steuert. Aus Sicht eines Hardwareentwerfers ist dies unter Umständen nicht immer erwünscht, da die Transparenz zwischen VHDL und entstehender Netzliste leidet. Um ein Systemverhalten realisieren zu können, das aus mehreren Operationen besteht, müssen diese entweder alle in einem Prozess zusammengefasst werden,

z.B. durch Verzweigungen oder Schleifen, oder es muss eine aufwendige Synchronisation zwischen mehreren Prozessen, die jeweils eine Operation bilden, durchgeführt werden.

```
MultProc : process is
  begin
    wait until rising_edge(clk);
    if start = '1' then
      done <= '0';
      P <= (others => '0');
      for i in A'range loop
        wait until rising_edge(clk);
        if A(i) = '1' then
          P <= (P(6 downto 0) & '0') + B;
        else
          P <= P(6 downto 0) & '0';
        end if;
      end loop;
      done <= '1';
    end if;
  end process;
```

Abbildung 2.8: VHDL Prozess mit mehreren **wait**-Anweisungen, der einen sequentiellen Multiplizierer realisiert [AL08].

Ein Beispiel aus [AL08] verdeutlicht diese Beschreibungsmöglichkeit. In Abb. 2.8 ist der VHDL-Quelltext eines sequentiellen Multiplizierers dargestellt, der eine **wait**-Anweisung innerhalb einer **for**-Schleife enthält. Der Multiplizierer besitzt zwei 8-Bit-Eingänge A und B und das Signal *start*, welches den Beginn der Operation auslöst. Das Ergebnis P der Multiplikation enthält ebenfalls 8 Bit. Die Operation findet damit in der Menge der ganzen Zahlen modulo 256 statt. Zusätzlich wird über das Signal *done* angezeigt, dass die Operation fertiggestellt wurde. Die Anzahl der Zeitschritte der Operation entspricht damit der Anzahl der Bits in A. Wenn die Berechnung beendet ist, wird der Gesamtprozess in einer Schleife ausgeführt, sodass die **wait**-Anweisung zu Beginn des Prozesses die Leerlaufoperation repräsentiert. Bei der Synthese dieser Beschreibung entsteht ein endlicher Automat, der abhängig vom konkreten Synthesewerkzeug mindestens ebensoviele Zustände haben wird wie die innere Schleife Durchläufe besitzt.

BlueSpec

Die Hardwarebeschreibungssprache Bluespec ist ebenso eine Alternative zum herkömmlichen Entwurf auf RT-Ebene [HA04]. Sie besteht aus einzelnen Regeln, die wiederum eine Bedingung und verschiedene Aktionen besitzen. Eine Regel kann immer dann aktiviert werden, wenn ihre Bedingung wahr wird. Bei mehreren Regeln die gleichzeitig aktiviert werden dürfen, muss nichtdeterministisch eine von ihnen ausgewählt werden. Die Aktionen einer Regel können den Zustand von speichernden Elementen verändern, indem sie ein Register beschreiben, einen Wert in eine Arrayzelle schreiben oder einem FIFO übergeben. Die zu schreibenden Werte und die Bedingungen der Regeln können wiederum den Inhalt der Speicherelemente lesen und in beliebigen Operationen verarbeiten.

Bei der Synthese eines solchen Systems muss bei der Auswahl der zu aktivierenden Regeln eine Arbitrierung stattfinden, sodass in jedem Systemtakt nur genau eine Regel ausgeführt wird. Dadurch ist jede Regel elementar und unabhängig vom restlichen Systemverhalten. Diese Unabhängigkeit ist einer der großen Vorteile der Methodik, da sie den Entwerfer davon entlastet, sich um die Koordination der Interaktionen zwischen verschiedenen parallelen Zustandsautomaten zu kümmern. Unter bestimmten Bedingungen, d.h. wenn zwei Regeln frei von Daten- und Schreibkonflikten sind (*read-after-write*, *write-after-write*) können sie auch nebenläufig in einem Takt ausgeführt werden. Das Synthesewerkzeug der Sprache, der Compiler, hat die Aufgabe die Anzahl der parallel aktivierten Regeln zu maximieren, um ein effizienteres Design zu ermöglichen.

Im Gegensatz zu ITL wird bei Bluespec keine Vollständigkeitsprüfung durchgeführt, d.h. es wird nicht überprüft, ob in jeder Situation eine weitere Regel aktiviert werden kann, oder ob ein Deadlock eintritt. Wenn ein Speicherelement in einer Regel nicht verändert wird, geht das System von einem "Halten" des Wertes aus. Obwohl der Ansatz in [HA04] als *operation-centric hardware description* bezeichnet wird, ist er nicht mit den Operationen dieser Arbeit zu vergleichen. ITL-Operationen einer Komponente sind im Gegensatz dazu per Definition nicht nebenläufig, sondern jede Operation ist allein für das Systemverhalten während ihres Zeitabschnittes verantwortlich. Weiterhin beziehen sich Bluespec-Regeln immer nur auf einen Takt, d.h. sie werten den Zustand des Systems zu einem bestimmten Zeitpunkt aus und bestimmen daraus den darauffolgenden Zustand, üblicherweise im nächsten Takt. In ITL-Operationen hingegen wird der Systemzustand über mehrere Takte hinweg ausgewertet und die Ausgänge müssen für die gesamte Dauer der Operation bestimmt werden. Aus diesen Gründen sind beide Methodiken trotz des Bezugs zu Operationen nicht unmittelbar zu vergleichen.

3 Operationseigenschaften als vollständige Verhaltensbeschreibung

3.1 Allgemeines

Um die Synthese von Hardware aus Operationseigenschaften verstehen zu können, wird in diesem Kapitel ein Überblick über den Aufbau und die Funktionsweise von Operationseigenschaften gegeben. In dieser Arbeit beschränken wir uns auf die Beschreibung von Eigenschaften in InterVal Language (ITL). Der Ursprung dieser Art temporaler Eigenschaften liegt in der formalen Verifikation, der es damit möglich wurde, zum einen das Verfahren des Interval Property Checking (IPC) zu nutzen und zum anderen eine Vollständigkeitsprüfung durchzuführen [Bor09].

Wie bereits in Abschnitt 2.2.4 genauer dargestellt, ist IPC ist ein Model Checking Verfahren, bei dem das zu prüfende Design für eine bestimmte Anzahl Takte "abgerollt" wird, und zusammen mit der Eigenschaft an ein formales Beweiswerkzeug, wie beispielsweise einen Erfüllbarkeitsprüfer, übergeben wird. Wenn das Beweiswerkzeug nachweisen kann, dass die Eigenschaft auf dem abgerollten Design immer erfüllt ist, ist das Beweisziel erbracht. Anderenfalls muss ein Gegenbeispiel ermittelt werden, welches eine Folge von Eingangsdaten enthält, die die Eigenschaften verletzt.

Es stellt sich dabei die Frage, welche Startzustände man zu Beginn der Eigenschaft annimmt. Im schon länger bekannten Bounded Model Checking (BMC) [Cla+01] wird einfach der Startzustand des System genutzt. Damit können Fehler 1. Art (false positives) ausgeschlossen werden, jedoch ist die Aussagekraft auf Zustände beschränkt, die innerhalb einer festen Anzahl Takte vom Startzustand aus erreichbar sind. Wenn jedoch ein Gegenbeispiel gefunden wird, kann man sich sicher sein, dass es ein gültiges Gegenbeispiel darstellt. Sofern im anderen Fall allerdings keines gefunden wird, kann man noch nicht darauf schliessen, dass die Eigenschaft hält.

Im IPC wird dies umgangen, indem zu Beginn der Eigenschaft alle Zustände erlaubt sind. Da dies auch nicht erreichbare Zustände einschliesst, erhält man unter Umständen false positives, die durch aufwendige manuelle Erreichbarkeitsanalysen oder automatisierte Verfahren [Ngu+08] ausgeschlossen werden müssen. Auf der anderen Seite kann man sich sicher sein, dass

es kein gültiges Gegenbeispiel gibt, wenn der Erfüllbarkeitsprüfer ein Halten der Eigenschaft anzeigt.

Die Anzahl Takte, die ein Design im IPC "abgerollt" wird, ergibt sich aus der Länge der Eigenschaft. Das heisst, je länger die Eigenschaft ist, desto aufwendiger wird die formale Verifikationsaufgabe. Die genaue Länge einer Eigenschaft, die noch bewiesen werden kann, hängt von der Komplexität des Designs und der Eigenschaft ab. Schränkt eine Eigenschaft die Freiheitsgrade des Designs stärker ein, ist es in der Regel möglich auch längere Eigenschaften zu beweisen. Im Allgemeinen ist die Komplexität jedoch nur schwer abzuschätzen.

Die vollständige formale Verifikation besteht im Wesentlichen aus zwei Teilen. Um eine Komponente vollständig zu verifizieren, muss zum einen eine Menge von Eigenschaften entworfen werden, die jede für sich mit Hilfe des Designs bewiesen wird. Zum anderen muss formal geprüft werden, dass die Eigenschaften selbst jeden Aspekt des Designs abdecken. Dieser Beweis wird Vollständigkeitsprüfung genannt.

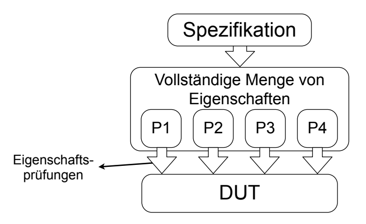


Abbildung 3.1: Vollständiger formaler Verifikationsfluss.

Man kann sich die vollständige formale Verifikation auch als eine Art Äquivalenzvergleich zwischen Design und Eigenschaftsmenge vorstellen. Die Eigenschaften müssen dabei ein Referenzmodell des Systems repräsentieren. Die Prüfung dieses Kriteriums übernimmt die Vollständigkeitsprüfung. Den eigentlichen Äquivalenzvergleich mit dem Design erledigt das Model Checking mit Hilfe von IPC, indem jede einzelne Eigenschaft auf dem Design erfüllt sein muss. In Abb. 3.1 ist dieser Ablauf schematisch dargestellt.

Da IPC schon kurz erklärt wurde, bleibt die Frage was während einer Vollständigkeitsprüfung passiert. Auf eine mathematische Formulierung soll an dieser Stelle verzichtet werden. In [Bor09] sind die entsprechenden Formeln ausführlich dargestellt. Ganz allgemein basiert die Prüfung darauf, dass das Verhalten des Systems in Operationen aufgeteilt wird. Jede dieser Operationen definiert das gesamte Verhalten während eines beschränkten Zeitabschnitts. Durch Aneinanderreihen verschiedener Operationen und durch

Prüfen der Übergänge zwischen diesen ist automatisch das Gesamtverhalten beschrieben. Eine Operation wird durch exakt eine temporale Eigenschaft abgebildet. Diese Eigenschaften werden dann Operationseigenschaften genannt.

Um den Übergang zwischen Eigenschaften zu gewährleisten, werden sogenannte *conceptual states* als wichtige oder konzeptionelle Systemzustände definiert. Jede Eigenschaft muss in einem solchen Zustand starten und enden. Gleichzeitig stellen diese Zustände sicher, dass die Eigenschaft auf dem Design nur ausgehend von diesem Startzustand geprüft wird.

Aus der Definition der Operationen als Beschreibung des Komponentenverhaltens während eines festen Zeitabschnitts lässt sich ableiten, dass die Modellierung mittels Operationseigenschaften nicht für jede Art Komponente geeignet ist. So ist es sehr umständlich parallele Funktionalität zu modellieren. Ein System muss deshalb in mehrere Blöcke aufgeteilt werden, die einzeln mit Operationseigenschaften beschrieben werden können. Die so entstandenen Blöcke werden Cluster genannt und müssen nicht den tatsächlichen Elementen der Hardware-Hierarchie entsprechen.

Zur Beschreibung eines Clusters ist auch die Modellierung des Datenflusses notwendig. Aus diesem Grund gibt es ein Datentypensystem für alle internen Signale, Ein- und Ausgänge einer Komponente, sowie für die temporären Objekte als Ergebnis der ITL-Operatoren. Die Datentypen sind an typische Hardwarebeschreibungssprachen angelehnt, wurden in ITL jedoch signifikant vereinfacht, sodass man im Wesentlichen nur zwischen sieben Datentypen unterscheidet:

- **unsigned** vorzeichenlose Bitvektoren,
- **signed** vorzeichenbehaftete Bitvektoren,
- **boolean** logische Werte,
- **bit** einzelne Bits,
- **array** Vektoren anderer Datentypen als **bit**,
- **record** Verbunddatentypen und
- **enum** Aufzählungstypen.

Die sehr strenge Typisierung von beispielsweise VHDL wird dadurch stark gelockert und die Beschreibung vereinfacht sich.

Für den in dieser Arbeit vorgestellten Synthesefluss wurde das ITL Typsystem übernommen, jedoch noch weiter vereinfacht. Das Typsystem von *vhisyn* kennt genau zwei Datentypen, den vorzeichenlosen (**unsigned**) und den vorzeichenbehafteten (**signed**) Bitvektor. Der logische Datentyp (**boolean**) und

das einzelne Bit (**bit**) werden beide auf einen vorzeichenlosen Bitvektor der Länge 1 abgebildet. Die drei Typen **array**, **record** und **enum** werden in der aktuellen Version überhaupt nicht unterstützt. Ein weiterer Unterschied ist die weniger strenge Typprüfung als in ITL. Die Datentypen der Argumente eines Makros werden in ITL explizit angegeben. Das Synthesewerkzeug *vhisyn* ignoriert diese Angaben und ermittelt die Datentypen aus den übergebenen Werten. Diese Typinferenz der Makroargumente unterscheidet sich vom originalen ITL. Eine synthetisierte Beschreibung wird sich jedoch genauso verhalten wie es der offiziellen ITL-Semantik entspricht, wenn die Eingangsbeschreibung korrektes ITL darstellt. Falls die Beschreibung nicht korrekt ist, wird das Synthesewerkzeug diese Art von Fehler nicht erkennen. Im Gegensatz dazu könnte ein Werkzeug, das die Typprüfung von ITL strikt einhält, den Nutzer darauf hinweisen.

In den folgenden Unterkapiteln werden nun der Aufbau und die Struktur der Operationseigenschaften genauer beleuchtet. Die Erläuterungen geben nur einen Überblick und sind oft nicht vollständig und in ihrer grammatikalischen Bedeutung präzise ausgedrückt. Der Fokus liegt in dieser Arbeit darauf, die notwendigen Grundlagen zu schaffen, um den Syntheseprozess in Kapitel 4 zu verstehen. Für weiterführende Information sei auf [OSS09] verwiesen.

Weiterhin wird in diesem Kapitel auf die Vollständigkeit eines Satzes von Operationseigenschaften eingegangen und erläutert, welche Kriterien dazu erfüllt sein müssen und wie dieser Nachweis erbracht werden kann.

3.2 Struktur der Eigenschaften

Eine Eigenschaft in ITL betrachtet immer einen Zeitraum, der aus einer festen Anzahl Grundtakte des zu untersuchenden Systems besteht. Dieser Zeitraum startet immer zu einem beliebigen Zeitpunkt t, sodass alle Zeitpunkte die in der Eigenschaft benutzt werden, relativ zu t ausgedrückt werden. Der Zeitpunkt zwei Takte nach t wird demnach mit $t + 2$ bezeichnet. Die grundlegende Struktur einer ITL Eigenschaft ist in Abb. 3.2 gegeben. Sie besteht aus mehreren Abschnitten von denen die Abbildung allerdings nur diejenigen darstellt, die in der vorliegenden Arbeit von Bedeutung sind.

3.2.1 Zeitvariablen

Der erste Abschnitt wird mit **for timepoints** eingeleitet und enthält eine Menge von Konstanten die Kurzbezeichner für die relevanten Zeitpunkte der Eigenschaft festlegen. Dadurch kann die Eigenschaft selbst wesentlich übersichtlicher und kompakter dargestellt werden. Weiterhin können im späteren

```
property Eigenschaftsname is
  for timepoints:
    t_start = t+2,
    t_end   = t_start+2;

  freeze:
    Freeze1 = Ausdruck @ t+1;

  assume:
    at t_start   : Annahme1;
    at t_start+1 : Annahme2;

  prove:
    at t_start+1                 : Zusicherung1;
    during [t_start+1,t_end]     : Zusicherung2;

  left_hook:  t_start;
  right_hook: t_end;
end property;
```

Abbildung 3.2: Syntax einer ITL Eigenschaft.

Entwurfsverlauf diese Zeitvariablen zentral geändert werden, was die Fehleranfälligkeit des Quelltextes reduziert.

3.2.2 Freezevariablen

Ein weiteres Element einer Eigenschaft, welches der Übersichtlichkeit dient und in komplexen Eigenschaften viel wiederkehrende Schreibarbeit einspart, wird im Abschnitt **freeze** beschrieben. Es werden feststehende Ausdrücke zu einem bestimmten Zeitpunkt mit einem symbolischen Namen versehen. Diese Freezevariablen genannten Bezeichner können im weiteren Verlauf der Eigenschaft immer dann eingesetzt werden, wenn der bezeichnete Ausdruck zu diesem Zeitpunkt noch einmal benötigt wird. So kann beispielsweise ein Datenwort am Eingang der Komponente zu Beginn der Eigenschaft in der Freezevariable "gespeichert" werden, um dann in den Zusicherungen gegen Ende der Eigenschaft wieder verwendet zu werden. In der Definition einer Freezevariable können auch alle vorhergehenden Freezevariablen genutzt werden, sodass es möglich ist, komplexe mehrstufige Berechnungen komplett im

freeze-Teil der Eigenschaft durchzuführen und in den Annahmen oder Zusicherungen nur das Ergebnis zu verwenden.

3.2.3 Annahmen und Zusicherungen

Der dritte, mit **assume** eingeleitete Abschnitt listet die Annahmen auf, die die Eigenschaft aktivieren können. Nur wenn jede einzelne dieser Annahmen erfüllt ist, ist die Eigenschaft aktiv und alle Zusicherungen müssen erfüllt werden. Der darauf folgende **prove** Teil legt die Zusicherungen fest, die für diese Eigenschaft gelten. Man kann sich den **assume** und **prove** Teil als eine einzige große Implikation vorstellen. Wenn man die Menge der Annahmen als $A_p = \{a_1, a_2, ..., a_n\}$ definiert und die Menge der Zusicherungen als $Z_p = \{z_1, z_2, ..., z_m\}$, ergibt sich die charakteristische Funktion der Gesamteigenschaft P zu

$$P = \bigwedge_{a_i}^{A_p} a_i \rightarrow \bigwedge_{z_i}^{Z_p} z_i.$$

Eine Annahme oder Zusicherung kann als temporaler Ausdruck bezeichnet werden und besteht wiederum aus zwei Teilen. Zum einen einer zeitlichen Angabe, die festlegt zu welchem Zeitpunkt, relativ zu t, der Ausdruck gilt. Wenn der Zeitpunkt mit **at** startet, betrifft dies einen einzelnen Zeitpunkt. Im Gegensatz dazu kann mit **during** ein Ausdruck für mehrere Zeitpunkte gelten. Dies ist gleichbedeutend mit einem zeitlichen Abrollen des Ausdrucks für alle Zeitpunkte in dem angegebenen Intervall entsprechend folgender Formel

$$during(t_1, t_2, expr) = \bigwedge_{i=t_1}^{t_2} at(i, expr).$$

Die Auswertung einer Annahme muss in einem Datentyp resultieren, der zu *wahr* oder *falsch* umgewandelt werden kann. Wenn die Eigenschaft in der Verifikation verwendet wird, gilt dies auch für Zusicherungen. In dem in dieser Arbeit vorgestellten Entwurfsprozess ist es jedoch erforderlich aus einer Zusicherung eine eindeutige Signalzuweisung zu extrahieren. Aus diesem Grund sollte eine Zusicherung immer in der Form

$$Signalname = Ausdruck$$

beschrieben werden. Wenn die betreffende Zusicherung in der formalen Verifikation verwendet wird, hat das Zeichen = die Bedeutung des Äquivalenzoperators, während es bei der Synthese für eine Wertzuweisung an ein Signal steht.

Der *Ausdruck* darf dabei nur von Signalwerten abhängen, die zu einem Zeitpunkt gelesen wurden, der vor dem Zeitpunkt der Zusicherung liegt. Dies stellt die größte Einschränkung von synthesefähigen Eigenschaften gegenüber der allgemeinen Form in der Verifikation dar. In Abschnitt 5.1.4 wird dieses Problem noch einmal ausführlicher untersucht.

3.2.4 Grenzen der Eigenschaft

Der letzte für die Synthese relevante Abschnitt einer Eigenschaft ist die Definition der Grenzen der Eigenschaft. Die Vollständigkeit eines Eigenschaftssatzes hängt davon ab, dass die Eigenschaften sequentiell verkettet werden können und dabei keine zeitlichen Lücken in der Beschreibung auftreten. Dazu ist es notwendig für jede Eigenschaft einen Beginn **left_hook** zu definieren, der angibt welches der erste Takt nach dem Ende der vorangegangenen Eigenschaft ist. Dieses Ende einer Eigenschaft wird durch den **right_hook** angegeben. Dabei ist zu beachten, dass der **right_hook** der einen Eigenschaft dem **left_hook** der darauf folgenden entspricht, sich diese also streng genommen um einen Takt überlappen. Jede Eigenschaft muss ihren jeweiligen **right_hook** festlegen und mit Ausnahme der Rücksetzeigenschaft, muss auch der **left_hook** definiert sein. Der Beginn der Rücksetzeigenschaft ist impliziert auf den Zeitpunkt t gelegt.

Die Festlegung eines **left_hook** bzw. **right_hook** in einer Eigenschaft schränkt nicht ein, zu welchen Zeitpunkten Annahmen und Zusicherungen definiert werden können. Es ist bspw. möglich und in vielen Fällen auch notwendig Zusicherungen zu einem späteren Zeitpunkt als dem **right_hook** zu definieren. Es gibt jedoch einige Einschränkungen, die die Synthese derartiger Eigenschaften betreffen (vgl. Kapitel 5).

3.2.5 Weitere Abschnitte

Weitere im Beispiel in Abb. 3.2 nicht enthaltene Abschnitte einer Eigenschaft wie bspw. **dependencies** oder **local_determination_requirements** sind wichtig, um in der formalen Verifikation den Vollständigkeitsbeweis erbringen zu können. So kann in den **dependencies** mit Hilfe von *constraints* angeben werden, welchen Einschränkungen die Eingänge der Komponente unterliegen, und es können *assertions* definiert werden, die Erreichbarkeitsinformationen des zu prüfenden Systems enthalten. Für eine Synthese, wie sie in dieser Arbeit durchgeführt wird, sind diese Abschnitte jedoch nicht von Bedeutung und werden deshalb hier nicht weiter beschrieben.

Die Ausdrücke die in Freezevariablen, Annahmen und Zusicherungen verwendet werden, können sowohl eine Vielzahl von arithmetischen, logischen

und weiteren Operatoren enthalten, als auch Signalnamen, die auf Eingänge, Ausgänge oder interne Signale verweisen. Für eine eingehende Erläuterung zu Syntax und Semantik von Ausdrücken sei an dieser Stelle erneut auf das Referenzhandbuch von ITL verwiesen [OSS09].

In Ausdrücken können so genannte Makros verwendet werden, welche in ITL das aus Programmier- und Beschreibungssprachen bekannte Konzept der Funktion repräsentieren. Im folgenden Abschnitt werden diese genauer untersucht, um die spätere Synthese besser zu verstehen.

3.3 Makros

Makros werden in ITL verwendet, um wiederverwendbare Funktionen zu beschreiben. Dies können zum einen allgemeine Funktionen sein, wie beispielsweise Wurzelberechnungen, Prüfsummenbestimmungen von Datenworten oder auch das einfache Heraustrennen von Teilvektoren aus größeren Vektoren. Zum anderen können Makros auch sehr spezielle Funktionalität einer Komponente beschreiben, wie die Berechnung spezieller arithmetischer Funktionen oder die Prüfung von Integritätsbedingungen eines Datenrahmens.

ITL kennt keine Anweisungen, wie man sie aus der imperativen Programmierung kennt, Makros werden ausschließlich funktional beschrieben. Dies bedeutet, dass der Funktionskörper eines Makros im Wesentlichen nur aus einem einzelnen Ausdruck besteht, der den Rückgabewert des Makros darstellt. Es gibt keine Seiteneffekte, da keine Zuweisungen an Signale oder Parameter stattfinden können. Dieser funktionale Beschreibungsstil ist für die Synthese von Hardware besonders gut geeignet. Jedes Syntaxelement kann direkt auf entsprechende Hardware-Operatoren abgebildet werden. Somit muss keine High-Level-Synthese stattfinden, wie man sie aus der Synthese von Sprachen wie C oder Matlab kennt. Nachteilig wirkt sich jedoch aus, dass die funktionale Beschreibung wesentlich weniger verbreitet unter Hardware-Entwerfern ist, als die klassische imperative Beschreibung, wie man sie aus VHDL oder Verilog Prozessen kennt. Insbesondere die Umwandlung von imperativen Programmschleifen in rekursive Funktionsaufrufe kann anfangs sehr umständlich wirken.

In ITL werden Makros vor allem zur Beschreibung des Datenflusses einer Komponente verwendet. Sie erleichtern die Beschreibung von ITL-Eigenschaften, indem mehrfach verwendeter Kode ausgelagert wird. Weiterhin erhöht sich das Verständnis der Eigenschaften, wenn komplexe Teilaufgaben in Makros stattfinden, anstatt den Quelltext der Eigenschaft unnötig zu vergrößern. Da ITL kein Konstrukt für Schleifen vorsieht, muss deren Ver-

halten durch rekursive Makroaufrufe realisiert werden. Dies entspricht dem üblichen Vorgehen in funktionalen Sprachen.

Im Wesentlichen kann jede kombinatorische Schaltung sehr direkt und effizient als funktionales Makro beschrieben werden. Es existieren in der Literatur Beispiele aus verschiedenen Anwendungsbereichen, die dies belegen. So können parallele Präfix-Schaltkreise[1] mit funktionalen Sprachen sehr elegant beschrieben und synthetisiert werden [She05].

Die grundlegende Syntax und Funktionsweise eines Makros wird in Abb. 3.3 anhand eines kleinen Beispielmakros zur Berechnung der ganzzahligen Quadratwurzel dargestellt. Der verwendete Algorithmus basiert auf [Rol87] und entspricht dem klassischen schriftlichen Wurzelziehen, nur zur Basis 2 anstatt 10. In Abb. 3.4 ist auch die entsprechende Funktion in C angegeben. Da ITL keinen Potenzoperator bereitstellt, muss auch dafür ein Makro bereitgestellt werden. An diesem Beispiel kann man erkennen, dass die iterative **while** Schleife in eine Rekursion umgewandelt wurde. Das ITL-Makro `sqrt_impl` ruft sich selbst für jede Iteration einmal auf, wobei die Variablen der C Funktion den Argumenten des Makros entsprechen.

3.3.1 Operandenwachstum

Was in diesem Beispiel nicht beachtet wird, ist die Besonderheit von ITL-Operatoren keinen Integer-Überlauf zuzulassen. So ergibt die Addition von zwei Bitvektoren im *unsigned* Format in ITL einen Bitvektor der ein Bit länger ist als der längere der beiden Operanden. Somit ist die Operation mathematisch immer korrekt, was das Schreiben von Eigenschaften für die formale Verifikation erleichtert. Auf der anderen Seite werden aber die Operanden in einem rekursiven Algorithmus mit jeder Addition oder Subtraktion immer länger, ohne dass dies wirklich notwendig wäre. Im Maro *sqrt_impl* wächst so die Bitbreite der Parameter *op* und *res* mit jeder Rekursion an und führt zu erhöhtem Berechnungsaufwand sowohl bei der Verifikation als auch bei der in dieser Arbeit durchgeführten Synthese. Da der Algorithmus ein Überlaufen der genannten Operationen ausschließt, kann durch explizites Abschneiden der unnötigen führenden Bits das Makro effizienter implementiert werden. Ein gutes Verifikations- oder Synthesetool sollte allerdings entweder in einem Vorverarbeitungsschritt oder während der Optimierung die unnötigen Bits, welche stets den Wert null annehmen, auch entfernen können.

[1] Ein paralleler Präfixschaltkreis ist ein kombinatorischer Schaltkreis, der n Eingänge $x_1, x_2, \ldots, x_n$ besitzt und daraus n Ausgänge $x_1, x_1 \circ x_2, \ldots, x_1 \circ \cdots \circ x_n$ berechnet, wobei $\circ$ eine beliebige assoziative, binäre Operation darstellt. Sie werden unter anderem für schnelle Addierer- oder FFT-Schaltkreise verwendet [Fic83].

```
macro sqrt(a : unsigned) : unsigned :=
  sqrt_impl(a,0,power2(a'length + a'length mod 2));
end macro;

macro sqrt_impl(op : numeric; res,one : unsigned)
    : unsigned :=
  if statically one > 0 then
    if op >= (res+one) then
      sqrt_impl(op-res+one,res div 2 + one,one div 4);
    else
      sqrt_impl(op,res div 2, one div 4);
    end if;
  else
    res;
  end if;
end macro;

macro power2 (i : numeric) : unsigned :=
  if statically i > 0 then
    power2(i-1) & "0";
  else
    1;
  end if;
end macro;
```

Abbildung 3.3: ITL-Makros zur Quadratwurzelberechnung.

3.3.2 Rekursionstiefe

Ein weiterer wichtiger Punkt beim Design von Makros ist die Rekursionstiefe. Ein Makro zum Umkehren der Reihenfolge von Bits in einem Vektor wird in einer simplen Implementierung ebenso viele Schritte benötigen wie die Länge des Vektors. In diesem einfachen Beispiel mag dies keine große Rolle spielen, falls jedoch sehr lange Bitvektoren in aufwendigeren Operationen verarbeitet werden müssen, ist diese Implementierungsvariante recht ineffektiv und es empfiehlt sich alternativ eine Teilung des Problems in zwei Hälften und rekursive Verarbeitung beider Teilprobleme. In Abb. 3.5 ist ein so entworfenes Makro zur Bitumkehrung dargestellt. In diesem wird statisch die Mitte des Vektors errechnet und dann zuerst die obere Hälfte umgekehrt. Danach

```
unsigned int sqrt(unsigned int num)
{
  unsigned int op = num;
  unsigned int res = 0;
  unsigned int one = 1 << 30;

  while (one != 0) {
    if (op >= res + one) {
      op -= res + one;
      res = (res >> 1) + one;
    }
    else
      res >>= 1;
    one >>= 2;
  }
  return res;
}
```

Abbildung 3.4: Mögliche C Implementierung zur Quadratwurzelberechnung.

wird die untere Hälfte umgekehrt und beide Ergebnisse wieder miteinander verkettet. Die Rekursionstiefe des so entstandenen Makros wächst nun nicht mehr linear mit der Länge des Vektors, sondern logarithmisch. Wenn allgemein wiederverwendbare Makros entworfen werden, ist dieses Vorgehen empfehlenswert, insbesondere wenn das Makro in einem Kontext aufgerufen wird, der sich bereits in einer sehr großen Rekursionstiefe befindet.

3.3.3 Sequentielle Makros

Bisher wurden nur kombinatorische Schaltungen betrachtet. ITL-Makros können jedoch auch sequentielle Schaltungen beschreiben, solange diese keine kombinatorischen Rückführungen enthalten. Die beiden zur Verfügung stehenden zeitlichen Operatoren sind *prev* und *next*. Sie lesen ein Signal zu einem Zeitpunkt, der eine bestimmte Anzahl Takte vor oder nach dem aktuellen Zeitpunkt liegt. Der *prev* Operator kann direkt auf ein gewöhnliches D-Flip-Flop abgebildet werden.

Durch rekursive Anwendung des *prev* Operators ist es weiterhin möglich, ein Signal über eine Dauer von mehreren Takten zusammenzusetzen und dann als kompletten Datenrahmen zu bearbeiten. Dies kann nützlich sein,

```
macro reverse (data : bit_vector) : bit_vector :=
  if statically data'length >= 2 then
    reverse(data(data'length div 2 - 1 downto 0))
      & reverse(data(data'high downto
                     data'length div 2));
  else
    data;
  end if;
end macro;
```

Abbildung 3.5: Makro zum Umkehren der Reihenfolge der Bits eines Vektors.

um beispielsweise eine Prüfsumme über einen Datenrahmen zu bilden, ohne in den Eigenschaften explizite Register zum Speichern der einzelnen Worte vorzusehen.

Im Falle des *next* Operators ist jedoch keine direkte Abbildung in Hardware möglich, da kein Grundelement existiert, welches zukünftige Werte eines Signals voraussagen kann. Durch Verschieben der zeitlichen Operatoren im Datenpfad (ähnlich Retiming in aktuellen Synthesewerkzeugen) kann eventuell ein *next* Operator mit Hilfe eines *prev* Operators aufgelöst werden. In Abschnitt 4.9.1 werden die Verfahren und auftretenden Probleme bei der Synthese von Makros in Register-Transfer-Level Hardware genauer untersucht.

3.4 Eigenschaftsgraph

Wie bereits erwähnt, müssen die Eigenschaften, um eine wirklich vollständige Beschreibung einer Komponente zu sein, zeitlich miteinander verknüpft werden. Für jede Eigenschaft darf es eine festgelegte Anzahl Nachfolger geben. Es muss jedoch mindestens einen Nachfolger geben, da sonst das weitere Verhalten der Komponente nicht festgelegt ist. Vom Entwerfer der Eigenschaften wird zu diesem Zweck ein Eigenschaftsgraph angelegt, der diese Vorgänger-Nachfolger-Beziehung spezifiziert. Es handelt sich dabei um einen zyklischen gerichteten Graph mit Schleifen,[2] jedoch ohne Mehrfachkanten. Es gibt immer genau eine Wurzel des Graphen, die die Reseteigenschaft repräsentiert und die ersten Takte der Komponente nach Eintreten der Resetbedingung (aktiviertes Reset-Signal) beschreibt. Der Graph ist von der Reseteigenschaft

[2] Als Schleifen bezeichnet man in der Graphentheorie Kanten, deren Start- und Endknoten gleich sind.

ausgehend stark zusammenhängend im Sinne der Graphentheorie, d.h. für jede Eigenschaft gibt es eine Folge von Eigenschaften, die von der Reseteigenschaft aus zu ihr führen.

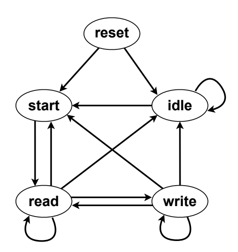


Abbildung 3.6: Eigenschaftsgraph einer Resampler Komponente.

In Abb. 3.6 ist ein solcher Eigenschaftsgraph für eine Resampler Komponente dargestellt [LPH10]. Die Reseteigenschaft *reset* kann von einer *idle* Eigenschaft oder einer *start* Eigenschaft gefolgt werden. Der Leerlauf des Systems wird dabei durch *idle* charakterisiert, während unter Annahme einer Startbedingung die Eigenschaft *start* aktiviert wird, die die Komponente in den eigentlichen Arbeitsmodus überführt. In diesem kann der Resampler nun in beliebiger Reihenfolge entweder Datensätze lesen (*read*) oder schreiben (*write*). Nachdem die zu bearbeitenden Datensätze erschöpft sind, kehrt das System wieder zurück in den Leerlaufmodus (*idle*) oder startet sofort die nächste Bearbeitungsrunde (*start*). Es ist dabei zu bemerken, dass der Eigenschaftsgraph einem Zustandsautomaten ähnelt, sich aber von ihm insofern unterscheidet, dass die Eigenschaften oft nicht nur einem Takt lang sind, wie die Zustandsübergänge eines klassischen Automaten. So ist in unserem konkreten Resampler Beispiel die *read*-Eigenschaft zwei Takte lang, da das Protokoll des gelesenen FIFO-Puffers eine zwei Takte lange Lesesequenz erfordert.

3.4.1 Der Eigenschaftsgraph als Kantengraph des Automaten der konzeptionellen Zustände

Es ist dabei anzumerken, dass der Eigenschaftsgraph an einen traditionellen Zustandsautomaten erinnert, der die Kontrollzustände des Systems beschreibt [HMU06]. Die Eigenschaften sind jedoch als Übergänge zwischen den charakteristischen Systemzuständen zu betrachten. Ihre Abarbeitung benötigt oftmals mehr als einen Takt. In manchen Anwendungsfällen kann die Länge der Eigenschaft bis zu mehreren hundert oder tausend Takten be-

tragen. Ein Beispiel ist die Verarbeitung von langen Datenrahmen im Telekommunikationsbereich. Jeder Rahmen kann durch eine einzige Eigenschaft beschrieben werden, wodurch eine intuitive und der originalen Spezifikation entsprechende Systembeschreibung entsteht. Die Systemzustände, wie bspw. der Start und das Ende eines Rahmens, werden in der formalen Verifikation konzeptionelle Zustände (*conceptual states*) genannt und ergeben sich implizit aus den Start- und Endzuständen der Eigenschaften. Man kann also den Eigenschaftsgraph als Kantengraph[3] des Zustandsautomaten der wichtigen Systemzustände ansehen.

Demnach ist es möglich den Eigenschaftsgraph aus den konzeptionellen Systemzuständen abzuleiten. Dazu müssen zunächst alle Systemzustände identifiziert werden. Diese können bspw. Beginn und Ende einer komplexen Busoperation oder mehrstufigen Berechnung sein. Aus diesen Zuständen kann nun ein Zustandsautomat für das System erzeugt werden. Jeder mögliche Übergang zwischen zwei Systemzuständen entspricht dabei einer Eigenschaft. Die Übergangsbedingungen repräsentieren die Annahmen der jeweiligen Eigenschaft, während die Zusicherungen den Ausgaben des Automaten entsprechen. Wenn ein solcher Zustandsautomat bereits vorliegt, kann durch Finden des entsprechenden Kantengraphen der Eigenschaftsgraph ermittelt werden. Ebenso kann aus dem Eigenschaftsgraph der entsprechende Zustandsgraph berechnet werden. Mit [Leh74] existieren Algorithmen für diese Aufgabe.

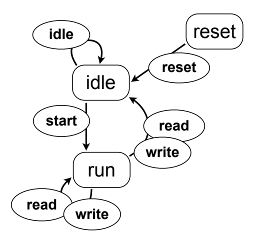


Abbildung 3.7: Möglicher Zustandsgraph einer Resampler Komponente.

Der Graph in Abb. 3.7 zeigt einen möglichen Graph der wichtigen Zustände des Resamplers. Die drei Zustände *reset*, *idle* und *run* repräsentieren dabei den Resetzustand, den Leerlaufzustand und den Betriebszustand. Die Kanten des Graphen entsprechen den Eigenschaften. Der Kantengraph dieses

[3] Der Kantengraph eines Graphen entsteht durch Vertauschung von Kanten und Knoten, wobei die Knoten im Kantengraph verbunden sind, wenn die entsprechenden Kanten im Originalgraph einen gemeinsamen Knoten besitzen [Die10].

Graphen entspricht im Wesentlichen dem Eigenschaftsgraph in Abb. 3.6. Es ist zu sehen, dass die Eigenschaften *read* und *write* an jeweils zwei Kanten annotiert sind. Dies ergibt sich aus der Tatsache, dass die Eigenschaft nach dem Verarbeiten des letzten Partikels wieder in den Leerlaufzustand übergeht, bei allen anderen Partikeln jedoch im Betriebszustand verbleibt. Weiterhin existiert im Zustandsgraph ein Übergang zwischen der *start* Eigenschaft und der *write* Eigenschaft, der im Eigenschaftsgraph nicht vorkommt.

3.4.2 Darstellung in ITL

Der ITL-Eigenschaftsgraph stellt diesen Kantengraph dar. Die konzeptionellen Zustände sind im Eigenschaftsgraph nicht explizit enthalten, sondern entsprechen seinen Kanten. Um in der formalen Verifikation dem Beweiswerkzeug den Nachweis einer Eigenschaft für das RT-Design zu ermöglichen, ist es deshalb nötig die konzeptionellen Systemzustände in den Eigenschaften zu verankern. Im Gegensatz dazu können ITL-Komponenten ohne Beschreibung der konzeptionellen Zustände in den Eigenschaften auskommen, wenn sie nicht für formale Verifikation sondern ausschliesslich für die Synthese verwendet werden. In Abschnitt 5.2.2 wird diese Tatsache aus Sicht der Synthese näher untersucht.

Neben den eigentlichen ITL-Eigenschaften muss in einer vollständigen Beschreibung auch ein Abschnitt **completeness** enthalten sein. Dieser besteht wiederum aus mehreren Teilen. Sie sind vorrangig für die formale Verifikation notwendig und werden deshalb an dieser Stelle nur auszugsweise beschrieben. So stellt der Teil **property_graph** die textuelle Beschreibung des Eigenschaftsgraphen dar. In Abb. 3.8 ist der den Eigenschaftsgraph betreffende Teil einer Vollständigkeitsbeschreibung dargestellt. Es werden nur die Kanten des Graphen beschrieben. Links des Pfeils befindet sich die jeweilige Vorgängereigenschaft und rechts des Pfeils der Nachfolger. Durch Auflistung von mehreren Eigenschaften auf einer oder beiden Seiten des Pfeils können entsprechend mehrere Kanten in einer Anweisung definiert werden. Dabei wird für jede Eigenschaftspaarung aus dem kartesischen Produkt der Mengen links und rechts des Pfeils eine Kante erstellt. So entspricht die Anweisung

```
reset, idle -> idle, start;
```

den vier Eigenschaftsübergängen

```
reset -> idle;
reset -> start;
idle -> idle;
idle -> start;
```

```
completeness resampler is
  ...
  reset_property:
    resampler_reset;

  property_graph:
    reset,idle -> idle,start;
    start -> read;
    write,read -> read,write,idle,start;
  ...
end completeness;
```

Abbildung 3.8: Eigenschaftsgraph des Resampler Beispiels in einer Vollständigkeitsbeschreibung.

Weitere nicht in der Abbildung dargestellte Abschnitte in der Vollständigkeitsbeschreibung sind die Aufzählung der Eingänge der zu verifizierenden Komponente und Angaben zur Determiniertheit der Ausgänge.

3.5 Beschreibung der Komponentenschnittstelle

Eine vollständige ITL-Eigenschaftsbeschreibung umfasst keine Definition der Ein- und Ausgangssignale einer Komponente mit Angabe des Datentyps. Weiterhin sind weder Parameter noch interne Signale, bzw. deren Datentypen vorgegeben. Bei einer formalen Verifikation der Eigenschaften in Bezug auf eine reale VHDL oder Verilog Implementierung extrahiert das formale Werkzeug diese Daten aus dem entsprechenden Entwurf. So sind in den Eigenschaften alle internen und Schnittstellensignale sowie Parameter mit dem korrekten Datentyp der Hardware verfügbar.

In einem eigenschaftsbasierten Entwurfsfluss existiert jedoch a priori keine Hardwarebeschreibung. Deshalb müssen die entsprechend benötigten Signaldefinitionen und Datentypen gesondert angegeben werden. Aus der Vollständigkeitsbeschreibung könnten die Namen der Ein- bzw. Ausgänge extrahiert werden, jedoch ohne Datentyp, was diese Information für eine Synthese unzureichend macht.

Um alle notwendigen Signale und Parameter und deren Datentyp nutzen zu können, wurde für diese Arbeit eine transparente Erweiterung von ITL vorgenommen. Die Vollständigkeitsbeschreibung wurde um einen Ab-

```
completeness resampler is
  ...
  /*structure:
    M         : generic unsigned(15 downto 0);

     in_state   : in unsigned(41 downto 0);
     in_pop     : out boolean;
     out_state  : out unsigned(41 downto 0);
     out_push   : out boolean;

     rd_count  : internal unsigned(15 downto 0);
     rand      : internal unsigned(23 downto 0);

     randgen  : component random(WIDTH=>24,SEED=>4000);
     randgen/dout -> rand;
     ...
  structure*/
  ...
end completeness;
```

Abbildung 3.9: Strukturbeschreibung in einer Vollständigkeitsbeschreibung.

schnitt **structure** ergänzt, der syntaktisch von ITL als Kommentar angesehen wird, von dem in dieser Arbeit vorgestellten Entwurfsfluss jedoch erkannt wird. Abb. 3.9 zeigt eine solche Strukturbeschreibung, die wiederum aus dem Resampler-Beispiel extrahiert wurde. Zu dieser Beschreibung gehören mehrere Arten von Anweisungen, die im Folgenden näher aufgeführt sind.

3.5.1 Parameter

Parameter (in VHDL als *generics* bekannt) sind Konstanten des Systems, die schon zur Synthesezeit feststehen. Sie dienen unter anderem der Erstellung von wiederverwendbaren Komponenten, da eine neue Konfiguration nur das Ändern des Parameters auf hoher Ebene und eine erneute Synthese nach sich ziehen. In der beschriebenen ITL Erweiterung werden Parameter durch einen Bezeichner, das Schlüsselwort **generic** und einen Datentyp definiert.

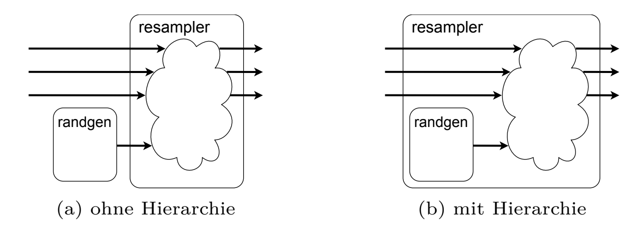
3.5.2 Signale

Eingänge bzw. Ausgänge werden ähnlich den Parametern durch einen Bezeichner, die Schlüsselworte **in** bzw. **out** und wiederum einen Datentyp festgelegt. In der Typdefinition eines Signals können dabei vorher definierte Parameter und andere konstante Ausdrücke benutzt werden. Man erhält somit parametrisierbare Signaldatenbreiten. Die Beschreibung von internen Signalen unterscheidet sich von den Schnittstellensignalen allein durch die Verwendung des **internal** Schlüsselworts.

3.5.3 Hierarchie

Der rekursive Aufbau von Hardwaresystemen aus kleineren Funktionsblöcken ist im Entwurf ein unabdingbares Funktionsmerkmal einer Beschreibungssprache. Oftmals wird dies durch Definition von Komponenten und deren Instanziierung in übergeordneten Modulen erreicht. ITL sieht eine solche rekursive Schachtelung von Modellen nicht vor und benötigt diese im Grunde auch nicht, um eine formale Verifikation durchzuführen. Die Festlegung der Vollständigkeits-Cluster kann völlig losgelöst von der Hierarchie der Hardwarebeschreibung sein. Wenn eine Eigenschaftsbeschreibung jedoch zum Entwurf von Komponenten (und nicht zur formalen Verifikation) verwendet werden soll, muss die Möglichkeit einer hierarchischen Struktur vorgesehen werden. In der beschriebenen erweiterten ITL-Syntax wird dies in den **structure** Abschnitt eingebettet. Man kann darin mit dem Schlüsselwort **component** eine andere Komponente im aktuellen Modul instanziieren. Dies kann entweder eine weitere vollständige ITL-Beschreibung sein oder auch ein bereits existierendes VHDL-Modul. So können beispielsweise Speichermodelle oder Cores zur effizienten Berechnung von arithmetischen oder trigonometrischen Funktionen in ein Eigenschaftsmodell eingebunden werden. Der Name der Instanz steht vor dem Doppelpunkt, während der Modulname der Unterkomponente dem Schlüsselwort **component** folgt. Die Unterscheidung zwischen Modul- und Instanzname erfolgt dabei analog zu VHDL.

Die Einführung von Komponenteninstanziierungen innerhalb einer ITL-Beschreibung ändert strukturell nichts an dem zu synthetisierenden Modell. Es wird lediglich eine Verschachtelung der Modelle vorgenommen, um wiederverwendbare Entwürfe in mehreren Ebenen zu erlauben. Ohne Hierarchie würde man nichts an Ausdrucksstärke verlieren, aber viel an Übersichtlichkeit. In Abb. 3.10 ist die Struktur des Resamplers schematisch abgebildet. Dabei entspricht Abb. 3.10(b) der textuellen Beschreibung in Abb. 3.9. Die so dargestellte Verschachtelung ist auch in mehrstufiger Form möglich.



(a) ohne Hierarchie (b) mit Hierarchie

Abbildung 3.10: Struktur des Resamplers und der Unterkomponente *randgen* mit und ohne Hierarchie.

3.5.4 Externe VHDL-Modelle

Wenn die instanziierte Komponente allerdings eine existierende VHDL-Beschreibung ist, muss diese zusätzlich dem Synthesewerkzeug bekannt gemacht werden. Dies geschieht durch die Definition einer **completeness** Anweisung, die nur einen einzelnen **structure** Abschnitt enthält, der die nach außen sichtbare Schnittstelle der VHDL-Komponente beschreibt. Zusätzlich muss jedoch noch eine Zeile

reference: external;

in die **completeness** Definition eingefügt werden. Diese Zeile macht den Syntheseprozess darauf aufmerksam, dass sich hinter diesem Modul eine VHDL-Beschreibung versteckt, die damit entsprechend im generierten Kode instanziiert wird.

3.5.5 Verknüpfungen von Signalen

Um hierarchische Beschreibungen effektiv nutzen zu können, müssen die Signale zwischen zwei Unterkomponenten oder zwischen Unterkomponente und Elternmodul miteinander verknüpft werden. Dies könnte in den Eigenschaften des Moduls durch entsprechende Zusicherungen geschehen, die zu jedem Zeitpunkt sicherstellen, dass das korrekte Signal zugewiesen wird. Der dadurch entstehende Aufwand wäre allerdings immens. Daher bietet es sich an, im übergeordneten Modul direkte Verbindungen zwischen Signalen zu ermöglichen. Dies kann wie in Abb. 3.9 dargestellt geschehen. Es muss nur das treibende Signal, ein `->` und das getriebene Signal als einzelne Anweisung in den **structure** Abschnitt aufgenommen werden. Die Identifizierung von Signalen der Unterkomponente geschieht dabei durch eine Qualifizierung

mittels des Instanznamens, der mit einem Schrägstrich getrennt vor dem Signalnamen steht.

3.6 Vollständigkeitsprüfung

Wie bereits zu Beginn dieses Kapitels erwähnt, kann eine Menge von Eigenschaften zur vollständigen formalen Verifikation eines Hardwareentwurfs eingesetzt werden. Ziel dieses Abschnitts ist es genauer zu beleuchten, wann eine solche Menge von Eigenschaften als vollständig bezeichnet werden kann. In [Bor09] werden die entsprechenden Kriterien mathematisch exakt dargestellt. Deshalb soll in dieser Arbeit nur ein informeller Überblick gegeben werden.

3.6.1 Vollständigkeit

Der Grundgedanke hinter der Vollständigkeit von Eigenschaften ist die exakte Abbildung jeder Folge von Eingangsvektoren des Designs auf eine eindeutige Folge von Ausgangsvektoren. Wenn diese Abbildung nachgewiesen werden kann, ist das Verhalten einer Komponente vollständig beschrieben. Die Erfüllung dieses Kriteriums ist allerdings nur durch einen Umweg zu erreichen. So wird in einem ersten Schritt sichergestellt, dass jede gültige Eingangsfolge eindeutig auf eine Folge von Eigenschaften abgebildet werden kann. In einem weiteren Schritt muss infolgedessen geprüft werden, ob jede Folge von Eigenschaften auch die Signalwerte an den Ausgängen des Designs eindeutig bestimmt.

3.6.2 Fallunterscheidungstest

Der erste Schritt, wird durch zwei Tests nachgewiesen. Zum einen existiert ein Fallunterscheidungstest (case split test), der feststellt, ob für jede Vorgängereigenschaft und Eingangsbelegung eine Nachfolgereigenschaft bestimmt ist. Wenn dieser Test für jede Eigenschaft erfüllt ist, kann man sicher sein, dass für jede Folge von Eingängen eine Folge von Eigenschaften existiert.

Der von den Onespin-Werkzeugen verwendete Fallunterscheidungstest prüft, ob *mindestens* eine Nachfolgereigenschaft existiert. Dadurch gelten auch Eigenschaftssätze als vollständig, in denen nebenläufig mehrere Eigenschaften aktiv sein können. Wenn diese parallelen Eigenschaften sich widersprechendes Ausgabeverhalten aufweisen, kann kein zugehöriger Schaltkreis existieren, sodass dieser Fall nicht betrachtet werden muss. Wenn jedoch alle parallelen Ausführungsstränge exakt das selbe Verhalten beschreiben, liegt

zwar ein bestimmtes Maß an Redundanz vor, was aber aus Verifikationssicht kein Problem darstellt.

Bei der Synthese eines Schaltkreises ist diese Nebenläufigkeit nicht erwünscht. Insbesondere bei der in Abschnitt 4.7 vorzustellenden Erzeugung eines deterministischen Automaten muss das Verhalten durch exakt eine Nachfolgereigenschaft bestimmt sein. Aus diesem Grund wird im weiteren Verlauf der Arbeit davon ausgegangen, dass der Fallunterscheidungstest derart durchgeführt wird, dass jede Eigenschaft unter jeder möglichen Belegung von Eingangssignalen und Zustandsvariablen *genau* eine Nachfolgereigenschaft besitzt.

3.6.3 Nachfolgertest

Zusätzlich muss jedoch ein sogenannter Nachfolgertest durchgeführt werden. Dieser prüft, ob eine Nachfolgereigenschaft ihre Aktivierungsbedingungen (Annahmen) nur aus den Werten der Eingänge herleitet oder aus Signalwerten, die von der Vorgängereigenschaft eindeutig festgelegt wurden. Der Test prüft demnach, ob zwei aufeinanderfolgende Eigenschaften "zusammenpassen". Zusammen mit dem Fallunterscheidungstest stellt der Nachfolgertest sicher, dass jede Eingangsfolge in exakt einer Folge von Eigenschaften mündet.

Die Unterscheidung, welche Signale bei diesen Tests die Eingangssignale darstellen, wird im Abschnitt **inputs** der Vollständigkeitsbeschreibung durch Auflistung der entsprechenden Signalnamen durchgeführt. Die Eingänge sind dabei nicht immer tatsächliche Eingänge des zu prüfenden Designs, sondern es können auch interne Signale verwendet werden. Damit ist man bei der Bestimmung der Verifikationscluster unabhängig von der tatsächlichen Hierarchie des Designs und kann die Grenzen des Clusters beliebig definieren.

3.6.4 Determinierungstest

Der dritte Test nennt sich Determinierungstest und stellt sicher, dass in jeder Folge von Eigenschaften jedes Ausgangssignal "determiniert" ist. Das bedeutet, dass sich in jeder Situation ein eindeutiger Signalwert ergibt. Das Verhalten des Modells ist somit deterministisch und läßt keine Freiheitsgrade an den Ausgängen zu. Es ist dabei zu beachten, dass entsprechend der Spezifikation vom Ersteller des Modells vorgegeben wird, welches die zu determinierenden Ausgangssignale sind und in welcher Form die Determinierung sichergestellt werden muss.

Die entsprechenden Determinierungsbedingungen werden im Abschnitt **determination_requirements** der Vollständigkeitsbeschreibung aufge-

führt. Es gibt zwei Formen der Determinierung. Zum einen kann ein Signal vollständig determiniert sein, was bedeutet, dass der Wert zu jeden Zeitpunkt bestimmt ist. Dies wird durch die Zeile

```
determined(Signalname), end_offset=Verzögerung;
```

ausgedrückt. Die Angabe **end_offset** kennzeichnet eine verschobene Determinierung, gegenüber den Grenzen der Eigenschaft. Bei der Modellierung von Pipelines vereinfacht diese Angabe die Durchführung des Determinierungstests.

Die zweite Form der Determinierung kommt zum Einsatz, wenn der Wert eines Ausgangssignals nur von Bedeutung ist, wenn eine bestimmte Bedingung eintritt. So hängt beispielsweise die Determiniertheit des Datenausgangs eines Speichers evtl. von der Aktivierung eines Lesesignals ab. Die Syntax dieser Form wird mit dem Schlüsselwort **if** eingeleitet und sieht wie folgt aus:

```
if (Bedingung) then
  determined(Signalname);
end if, end_offset=Verzögerung;
```

3.6.5 Resettests

Zusätzlich zu diesen drei Tests müssen für den Resetfall abgewandelte Formen der anderen Tests genutzt werden. Diese als Resettests bezeichneten Prüfungen beweisen somit die entsprechenden Fragestellungen vor dem Hintergrund der speziellen Gegebenheiten der Reseteigenschaft.

3.6.6 Trivialdesign

Die Beweisführung der Tests hängt nicht von einem tatsächlichen Hardwareentwurf ab. Sie werden vielmehr auf einem "trivialen" Design durchgeführt, welches zwar alle Signale und deren Datentypen kennt, jedoch kein weiteres Verhalten implementiert. Die Ausgänge und internen Signale besitzen im Trivialdesign keinen Treiber, d.h. sie können zu jedem Zeitpunkt einen der Werte ihres Wertebereichs frei auswählen.

Demnach kann eine Menge von Eigenschaften auf Vollständigkeit geprüft werden kann, ohne dass bereits eine Implementierung vorliegt. Dies hilft dabei Fehler zu vermeiden und ermöglicht ein schnelleres Auffinden von noch nicht beschriebener Funktionalität während des Entwurfs der Eigenschaften. Bei der formalen Verifikation der Eigenschaften vor Erstellung oder automatisierten Generierung einer Implementierung muss jedoch das Trivialdesign

vorhanden sein. Es kann auf einfache Art und Weise aus der in Abschnitt 3.5 vorgestellten Schnittstellenbeschreibung der Komponente hergeleitet werden.

3.6.7 Konzeptionelle Zustände

Weiterhin ist zu erwähnen, dass die konzeptionellen Zustände des Designs in den Eigenschaften nicht gesondert markiert werden müssen. [Bor09] weist darauf hin, dass die Zustände in der Praxis oft keinem konkreten Zustand des Designs entsprechen, sondern auch von Eingangs- bzw. Ausgangssignalen abhängen können und im Grenzfall komplett ohne Nutzung interner Signale auskommen. Der hier beschriebene Entwurfsfluss macht sich dies zunutze, indem Eigenschaften ohne explizit definierte wichtige Zustände entworfen und formal verifiziert werden. Lediglich die in Abschnitt 6.1 vorgestellte formale Verifikation des generierten Entwurfs benötigt die Verknüpfung zwischen Eigenschaftsbeginn und -ende und den Zuständen des Designs.

Von den beschriebenen Tests ist vor allem der Nachfolgertest dafür zuständig, dass die Definition der konzeptionellen Zustände den korrekten Übergang von einer Eigenschaft zur nächsten gewährleistet.

3.6.8 Implementierbarkeit

Weiterhin ist zu betrachten, ob eine vollständige Menge von Eigenschaften überhaupt implementierbar ist. Es sei an dieser Stelle auf [Bor09, Seite 97] verwiesen:

> Der Vollständigkeitsbeweis für Operationsautomaten wird mit den selben Tests durchgeführt wie der Vollständigkeitsbeweis für Operationseigenschaften. [...] Auf diese Weise ist der Vollständigkeitsprüfer auch einsetzbar als Plausibilitätsprüfung für einen Operationsautomaten, für den es noch keine Implementierung gibt. Diese Plausibilitätsprüfung lässt die Frage der Implementierbarkeit des Operationsautomaten offen. Diese Frage klärt sich erst, wenn es zum Operationsautomaten eine Implementierung gibt, die gegen alle Transitionen des Operationsautomaten verifiziert wurde.

Der Ausdruck Operationsautomat entspricht dabei dem durch Eigenschaften und Eigenschaftsgraph spezifizierten Modell.

Demnach stellt der Determinierungstest nicht sicher, ob das durch die Eigenschaften spezifizierte Verhalten überhaupt realisierbar ist, d.h. ob es einen Schaltkreis geben kann, für den alle Eigenschaften erfüllt sind. Eine solche Situation kann auftreten, wenn zwei Eigenschaften parallel gestartet werden

und noch keine von ihnen durch die Annahmen ausgewählt wurde. Wenn in einer solchen Situation beide Eigenschaften widersprüchliche Zusicherungen aufweisen, kann es keinen Entwurf geben, der sie beide erfüllt. Der Entwurf kann nur eine der beiden Zusicherungen erfüllen. Die andere Zusicherung schlägt demnach bei der Eigenschaftsprüfung fehl.

Da in der formalen Eigenschaftprüfung aber ein implementiertes Design vorliegt, auf dem alle Eigenschaften erfüllt sind, muss der Determinierungstest die Realisierbarkeit nicht überprüfen. Wenn allerdings das Design erst aus den Eigenschaften synthetisiert werden soll, hat man die Realisierbarkeit nicht a priori gegeben, d.h. man weiß zur Designzeit nicht, ob es möglich sein wird, eine gültige Realisierung zu erstellen. Auf dieses Problem und die dabei angestrebte praktische Lösung wird im weiteren Verlauf der Arbeit genauer eingegangen.

Um das Beispiel zu illustrieren ist in Abb. 3.11 eine vollständige ITL-Komponente gegeben. In den beiden Eigenschaften a und b existiert jeweils die einzelne und entscheidende Annahme zum Zeitpunkt $t+2$, während die einzige Zusicherung zum Zeitpunkt $t+1$ stattfindet. Da ein System zu $t+1$ nicht wissen kann, wie sich das Eingangssignal im nächsten Zeitpunkt verhalten wird, ist dieses System nicht implementierbar. Der Vollständigkeitstest wird diese Komponente als korrekt ausweisen, während es kein Design geben kann, das beide Eigenschaften gleichermaßen erfüllt.

```
constraint no_rst := rst = 0;
end constraint;

property reset is
  dependencies: no_rst;
  assume:       reset_sequence;
  prove:        during [t,t+2] : out1 = 0;
  right_hook:   t+2;
end property;

property a is
  dependencies: no_rst;
  assume:       at t+2 : in1;
  prove:        at t+1 : out1 = 0;
  left_hook:    t;
  right_hook  : t+1;
end property;

property b is
  dependencies: no_rst;
  assume:       at t+2 : not in1;
  prove:        at t+1 : out1 = 1;
  left_hook:    t;
  right_hook:   t+1;
end property;

completeness not_implementable is
  inputs:                      rst, in1;
  determination_requirements: determined(out1);
  reset_property:              reset;
  property_graph:              reset -> a,b;
                               a,b -> a,b;
  /*structure:                 in1  : in boolean;
                               out1 : out boolean;
  structure*/
end completeness;
```

Abbildung 3.11: ITL-Quelltext einer nicht implementierbaren Eigenschaftsmenge, die jedoch der Vollständigkeitprüfung standhält.

4 Synthese von Operationseigenschaften

4.1 Überblick über den Synthesevorgang

In diesem Kapitel soll der eigentliche Synthesevorgang einer vollständigen Menge von Operationseigenschaften ausführlich beschrieben werden. Ziel ist dabei die Ausgabe einer Hardware-Beschreibung in VHDL auf Register-Transfer-Ebene. Dieses Kapitel ist wie folgt strukturiert: Nach einem kurzen Überblick sollen die beteiligten Algorithmen und die Erzeugung von passenden Modellen vorgestellt werden. Gegen Ende des Kapitels wird der Fokus mehr auf der konkreten Abbildung der Modelle in entsprechende Hardware-Strukturen liegen.

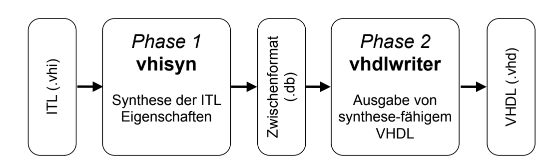


Abbildung 4.1: Grundlegender Entwurfsfluss aus Operationseigenschaften.

Wie in Abb. 4.1 dargestellt, besteht der Synthesevorgang im Wesentlichen aus zwei Phasen. In der ersten Phase werden die ITL-Dateien eingelesen und die entsprechende Hardware in einem Zwischenformat erzeugt. Das Zwischenformat wird dann in der zweiten Phase genutzt, um die zugehörige VHDL-Beschreibung zu erzeugen. Durch diese Trennung zwischen Syntheseprozess und VHDL-Generierung wird erreicht, dass zum einen weitere Werkzeuge, bspw. für Visualisierung, Simulation oder Debugging, direkt auf dem Zwischenformat aufsetzen können und nicht erst den erzeugten Quellkode parsen müssen. Zum anderen kann die Beschreibungssprache ausgetauscht werden kann, ohne die anderen Programmteile verändern zu müssen.

Das Zwischenformat besteht aus einer Datenstruktur, die eine allgemeine Form von hierarchisch aufgebauter Hardware widerspiegelt. Innerhalb dieser Hierarchie können allgemeine Basisblöcke verwendet werden, die den Operatoren von ITL entsprechen und auch deren semantisches Verhalten annehmen. In vielen Fällen stimmt das nicht mit dem üblichen Verhalten von

Hardwareoperatoren überein. So hat ein Block im Zwischenformat, der den ITL-Additionsoperator repräsentiert, immer ein "überlaufendes" Verhalten, welches bei der Abbildung auf VHDL entsprechend berücksichtigt werden muss.

Signale können vorzeichenbehaftete oder vorzeichenlose Bitvektoren sein. Innerhalb der hierarchischen Module werden sie mit Ein- oder Ausgangsports verknüpft, was entweder die Ports des aktuellen Elternmoduls oder die Ports der Basisblöcke sind. Verknüpfungen zwischen zwei Signalen sind nicht erlaubt.

Bei den hardwareerzeugenden Schritten des Synthesevorgangs in Phase 1 wird die Datenstruktur des Zwischenformats systematisch aufgebaut. Am Ende dieses Vorgangs ist die Struktur komplett und kann mit entsprechenden Visualisierungstools betrachtet werden. Im Rahmen dieser Arbeit ist ein solches Werkzeug entstanden, welches die gespeicherte Datenstruktur einliest und ähnlich der *schematics*-Darstellung in bekannten Simulations- bzw. Synthesetools darstellen kann. Im Abschnitt 6.6 wird das Werkzeug vorgestellt.

4.1.1 Details von Phase 1

In diesem Abschnitt soll ein grober Überblick über den Synthesevorgang in Phase 1 gegeben werden. Die im weiteren Verlauf folgenden Unterkapitel werden auf die einzelnen Punkte genauer eingehen.

1. Einlesen der ITL-Dateien und Überführung in einen abstrakten Syntaxbaum.

2. Aufbau der Modulhierarchie durch Auswertung des **structure** Abschnitts und Durchführung der Synthese für eventuelle Untermodule.

3. Erstellung des Eigenschaftsgraphen und Speichern der Graphenrepräsentation.

4. Auswertung und Hardwaregenerierung für alle im Eigenschaftsgraph enthaltenen Eigenschaften.

5. Konstruktion des Kontrollautomaten der Komponente basierend auf dem Eigenschaftsgraph.

6. Erzeugung der Ausgangslogik, die die Zusicherungen je nach Zustand des Kontrollautomaten an die Ausgänge weiterleitet.

7. Speichern der entstandenen Datenstruktur der Gesamthardware.

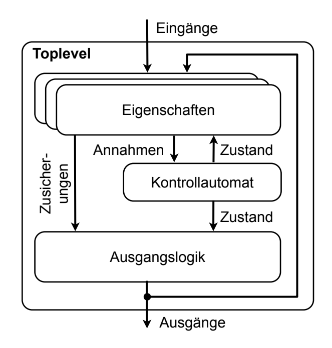


Abbildung 4.2: Aufbau eines Hardwaremodells, das aus einer vollständigen Eigenschaftsmenge erzeugt wurde.

Abb. 4.2 zeigt die Struktur eines Modells, das entsprechend dieser Punkte erzeugt wurde. Interne Signale werden im Gegensatz zu den Ausgangssignalen nicht mit den Ausgängen des Toplevels verbunden.

4.1.2 Details zu Phase 2

In der zweiten Phase wird die Repräsentation der entstandenen Hardware im Zwischenformat in synthesefähiges VHDL auf RT-Ebene transformiert. Das entsprechende Werkzeug *vhdlwriter* lädt das Zwischenformat, und übersetzt rekursiv jedes Modul und eventuell enthaltene Untermodule.

Da durch die Synthese von Makroaufrufen eine sehr große Anzahl von verschiedenen Modulen entstehen kann, wurde bei der Generierung die VHDL-Anweisung **block** verwendet. Es erlaubt die Definition einer hierarchischen Struktur, ohne dass eigenständige Entwurfseinheiten erstellt werden müssen.

Es entsteht nur eine einzige VHDL Ausgabedatei, die ein Modul (**entity** und **architecture**) für die Toplevel-Komponente beinhaltet. Alle tieferen Ebenen sind mit Blöcken realisiert.

4.2 Einlesen von ITL

ITL kann in zwei verschiedenen Sprachstilen verwendet werden. Die auf VHDL basierende Variante wird *vhi* genannt, was gleichzeitig die Endung der betroffenen Dateien darstellt. Die Verilog-Variante wird als *vli* bezeichnet. Der Unterschied ist rein syntaktischer Natur und betrifft die Operatoren, die in Ausdrücken verwendet werden können.

In dieser Arbeit wird nur die VHDL Sprachvariante genutzt, d.h. die Eingangsdaten des ersten Schrittes von *vhisyn* sind ITL-Dateien in *vhi* Syntax. Für jeden Synthesevorgang werden alle benötigten Dateien zu Beginn eingelesen und in einen abstrakten Syntaxbaum (AST) überführt. Die ITL-Grammatik wurde mit Hilfe von ANTLR [Par07] in einen Parser gewandelt, der dann aus *vhisyn* heraus aufgerufen wird. ANTLR hat einen eingebauten Mechanismus, um zu einer gegebenen Grammatik einen passenden AST zu erstellen.

Nachdem der AST aller ITL-Dateien vorhanden ist, werden die Konstrukte der obersten Ebene herausgesucht und in getrennten ASTs gespeichert. So werden alle Makros, Eigenschaften sowie Vollständigkeitsbeschreibungen extrahiert. Sie können dann im weiteren Verlauf der Synthese anhand ihres Namens direkt gefunden und evaluiert werden, ohne dass jedes Mal eine Suche in den ASTs aller Dateien angestossen werden muss.

Die Laufzeitkomplexität des mit ANTLR generierten LL(*) Parsers[1] ist linear bezüglich der Länge der einzulesenden Datei. Eine Ausnahme bildet das Parsen von Funktionsaufrufen, bei welchem die Option **backtrack=true** genutzt werden musste, da ANTLR ansonsten keinen Parser aufbauen konnte. Im speziellen Fall besitzt die Grammatikregel *callExpr* zwei nicht eindeutige Alternativen, die nachfolgend in Backus-Naur-Form (BNF) dargestellt sind:

$$
\begin{array}{lcl}
\textit{callExpr} & ::= & \textbf{ID}\ \textbf{"("}\ \textit{expression}\ (\ \textbf{","}\ \textit{expression}\ )*\ \ \textbf{")"} \\
 & | & \textbf{ID} \\
 & | & \ldots
\end{array}
$$

Durch die eingefügte Backtracking-Option testet der Parser zuerst die erste Alternative (Funktionsaufruf mit Argumenten), und erst wenn diese fehlschlägt, wird die zweite Alternative (alleinstehender Bezeichner oder Funktionsaufruf ohne Argumente) versucht. Da bei der öffnenden Klammer der ersten Alternative der Konflikt aufgelöst wird, ohne dass weitere Unterregeln mit möglicherweise komplexem Verhalten aufgerufen werden, führt das Backtracking nicht zu einer exponentiell komplexen Rekursion. Damit ist der entstandene Parser trotz dieser Ausnahme sehr schnell und kann ITL-Dateien in linearer Laufzeit einlesen.

[1] Ein LL(k) Parser verarbeitet die Eingabe von **L**inks nach rechts um eine **L**inksableitung zu berechnen. Der Parameter k gibt an, wie viele Token der Parser vorausschauen kann. Entsprechend kann ein LL(*) Parser unbegrenzt viele Token voraussehen. Nähere Informationen zu mit ANTLR erzeugten LL(*) Parsern sind in [Par07] nachzulesen.

4.3 Auswertung der Strukturbeschreibung

Aus der Menge der ASTs wird nun diejenige Vollständigkeitsbeschreibung herausgesucht, die das Toplevel der zu synthetisierenden Komponente darstellt. Die Festlegung welche Komponente als Toplevel betrachtet wird, geschieht dabei mit Hilfe der *vhisyn* Kommandozeilenoption -t. Zu Beginn wird ein leeres Modul in der Datenstruktur des Zwischenformats angelegt. Dieses wird dann in den nachfolgenden Schritten kontinuierlich gefüllt.

Danach werden alle im Abschnitt **structure** der Vollständigkeitsbeschreibung vorkommenden Anweisungen der Reihe nach abgearbeitet. Die folgenden Unterabschnitte betrachten jeweils eine dieser Anweisungen.

4.3.1 Signalverzeichnisse

Um Signale, die bereits einmal angelegt wurden, im späteren Verlauf wiederfinden zu können, gibt es ein lokales und ein globales Signalverzeichnis. Diese entsprechen im Wesentlichen den Symboltabellen aus traditionellen Programmiersprachen. Alle Parameter und global sichtbaren Symbole werden dabei im globalen Verzeichnis abgelegt, während die Argumente eines Funktionsaufrufs im lokalen Verzeichnis gespeichert werden. Das globale Verzeichnis ist für die gesamte Komponente, inklusive Eigenschaften und Makroaufrufe, gleich. Nur wenn eine Komponente Submodule instanziiert, besitzen diese ein neues globales Signalverzeichnis. So wird bei der Auswertung einer Eigenschaft oder eines Makros das globale Verzeichnis weitergenutzt, während das lokale Verzeichnis auf jeder Hierarchiestufe neu angelegt wird.

Globales Verzeichnis

Alle Signale, die im weiteren Verlauf der Auswertung komponentenweit unter ihrem Namen ansprechbar sein sollen, werden im globalen Signalverzeichnis gespeichert. Dies betrifft die Parameter, sowie die explizit definierten Signale der Komponente. Das Verzeichnis bildet den Namen auf ein entsprechendes Netz und eine konkrete Signalbelegung ab

$$globals : Name \rightarrow (\textit{Netz-ID}, \textit{Signalbelegung}).$$

Die *Signalbelegung* ist dabei ein Bitvektor der Werte 1,0 und $-$ (unbekannt). Der Bitvektor kann den Wert des Netzes im Zweierkomplement speichern. Zusätzlich wird noch abgelegt, ob die Signalbelegung vorzeichenbehaftet interpretiert werden soll oder nicht, bzw. wie die Bits des Vektors indiziert werden müssen, bspw. (31 *downto* 0) oder (8 *to* 23).

Wenn später ein beliebiger Ausdruck innerhalb der Komponente ausgewertet werden soll, wird ein im Ausdruck auftauchender Bezeichner zuerst im entsprechenden lokalen Verzeichnis gesucht. Falls dies nicht erfolgreich ist, wird die Suche im globalen Signalverzeichnis fortgesetzt. Wenn er auch dort nicht gefunden wird, muss ein Fehler signalisiert werden.

Ein besonderes Szenario tritt auf, wenn für einen Namen nur ein konstanter Wert benötigt wird. Dies kann unter anderem vorkommen, wenn die Zeitpunkte einer Eigenschaft ausgewertet werden. Sie müssen zu konkreten Ganzzahlen evaluieren und können innerhalb beliebig komplexer Ausdrücke auch Parameter benutzen. In diesem Fall wird kein Netz, sondern nur die Belegung des Eintrags im Signalverzeichnis benötigt. Damit wird keine tatsächliche Hardware bei der Auswertung generiert.

Im Fall einer nichtkonstanten Auswertung wird in der generierten Hardware die Netz-ID verwendet. Wenn das Netz in einem anderen Scope innerhalb des selben Moduls verwendet wird, muss es gegebenenfalls erst mit Hilfe zusätzlich eingefügter Ports auf die benötigte Ebene weitergeleitet werden. Dies betrifft zum Beispiel globale Eingänge, die innerhalb eines Makros direkt benutzt werden, ohne dass sie dem Makro als lokales Argument übergeben wurden.

Lokales Verzeichnis

Das lokale Signalverzeichnis bildet nicht nur einen Bezeichner, sondern auch einen Zeitpunkt auf ein konkretes Netz ab:

$$\mathit{locals} : \mathit{Name} \times \mathbb{Z} \rightarrow (\mathit{Netz\text{-}ID}, \mathit{Signalbelegung}).$$

Die Bedeutung von Netz-ID und Signalbelegung ist dabei die selbe wie im globalen Signalverzeichnis. Der Zeitpunkt im Argument dieser Abbildung gibt an, wann innerhalb einer Eigenschaft der betreffende Bezeichner genutzt werden soll. Bei Signalen die sich ausschließlich aus Ein- bzw. Ausgängen herleiten oder die mit Namen versehenen Zeitpunkte (**for timepoints**) einer Eigenschaft verwenden, spielt der Zeitpunkt der Auswertung keine Rolle. Lediglich bei Namen, die sich unter anderem aus den Freezevariablen herleiten, muss er beachtet werden. In Abschnitt 4.5.4, der sich mit Freezevariablen beschäftigt, wird dies detailliert erläutert.

4.3.2 Parameter

Die Definition der Parameter mit dem Schlüsselwort **generic** führt zur Erstellung eines Signals mit konstantem Wert. Der Wert selbst muss dabei von der übergeordneten Ebene übergeben worden sein. Im Falle einer Auswertung

auf Toplevel, werden alle Parameter per Kommandozeilenoption -g übergeben. Der Nutzer kann die Option erneut für jeden Parameter setzen. Sie verlangt je zwei Argumente, zum einen den Namen des Parameters und zum anderen den Wert in Ganzzahlnotation.

Wenn also die zu synthetisierende Komponente den Parameter **WIDTH** unterstützt, muss *vhisyn* auf der Kommandozeile ein Argument der Form

$$-g \ \mathbf{WIDTH} \ 32$$

übergeben werden. Der so übergebene Wert wird entsprechend dem Datentyp des Parameters aus der **generic** Anweisung interpretiert und einem neu generierten Signal als Konstante zugewiesen.

Wenn die Komponente allerdings nicht als Toplevel, sondern auf einer niedrigeren Hierarchiestufe ausgewertet wird, muss der Wert des Parameters von der übergeordneten Hierarchieebene übergeben worden sein. Es ist keine Möglichkeit vorgesehen Defaultparameter zu verwenden. Die zugehörige Definition in der **component** Anweisung wird in Abschnitt 4.3.4 beschrieben. Die entstehende Hardware im Zwischenformat führt keine Übergabe des Parameters durch, sondern der Wert wird in der Unterkomponente als neue Konstante definiert.

4.3.3 Signale

Für alle Eingänge, Ausgänge und internen Signale wird im Zwischenformat ein entsprechendes Netz vom passenden Datentyp angelegt. Bei Ein- und Ausgängen muss zusätzlich noch ein Port in der Modulschnittstelle erstellt werden, der mit dem vorher angelegten Netz verbunden wird. Weiterhin werden die Netze unter ihrem Namen im globalen Signalverzeichnis abgelegt. Die gespeicherte Signalbelegung hat dabei zwar den korrekten Typ, ist aber mit unbekannten Bits ($-$) gefüllt.

4.3.4 Untermodule

Jede **component** Anweisung führt zur Erstellung und Instanziierung eines Moduls auf der nachfolgenden Hierarchieebene. Dabei werden die folgenden vier Schritte durchgeführt:

1. *Auswertung der Parameterausdrücke im lokalen Scope.*
   In diesem Schritt wird die konkret benötigte Parameterkonfiguration für das Submodul berechnet. Dazu werden alle Ausdrücke ausgewertet, die in der Parameterliste der **component** Anweisung aufgeführt werden. Jeder Ausdruck muss zu einem konstanten Wert evaluieren.

Die so ermittelte Parameterkonfiguration entspricht dem Vorgang aus Abschnitt 4.3.2 (Übergabe der Toplevel-Parameter mit der Option $-g$) auf allen weiteren Hierarchiestufen.

2. *Generierung des Submodulnamens.*
   Da Untermodule mit unterschiedlichen Parametern verwendet werden können und diese Fälle nicht nur zu unterschiedlichen Instanzen sondern zu komplett unterschiedlichen Moduldefinitionen führen müssen, ist es erforderlich das Modul bezüglich der verwendeten Parameterkonfiguration eindeutig zu machen. Dazu wird an den Modulnamen die Liste der Parameterwerte angehängt. Aus der folgenden Komponenteninstanziierung, in der die drei Parameter M, $SWIDTH$ und $MWIDTH$ auf die Werte 8192, 14 und 32 gesetzt werden

   ```
   rs : component resampler(M      => 8192,
                            SWIDTH => 14,
                            WWIDTH => 32);
   ```

   ensteht damit ein Modul mit dem Namen

   ```
   resampler_M_8192_SWIDTH_14_WWIDTH_32.
   ```

   Jede Instanziierung einer Komponente, die eine unterschiedliche Parameterkonstellation verwendet, erzeugt damit ein eigenes Modul.

3. *Rekursive Ausführung des gesamten Entwurfsflusses für das Submodul.*
   Sofern das Submodul in exakt der gleichen Parameterkonfiguration noch nicht erstellt wurde, wird es nun beginnend mit der Auswertung seiner Strukturbeschreibung generiert. Falls das Submodul mit diesen Parametern in der momentanen Synthese bereits vorher benutzt wurde, kann man es wiederverwenden und es wird direkt mit dem nächsten Schritt fortgefahren.

4. *Instanziierung des Submoduls im aktuellen Modul.*
   Von dem neu erstellten oder wiederverwendeten Submodul wird eine Instanz im aktuellen Modul angelegt. Der Name der Instanz ist Teil der **component** Anweisung (Bezeichner rs im obigen Beispiel) und wird direkt verwendet. Zusätzlich für jeden Ein- oder Ausgang der neuen Instanz wird ein neues Netz angelegt und mit dem Port verknüpft. Dadurch kann beim späteren Anlegen von Verbindungen zu diesen Schnittstellensignalen schon auf ein bestehendes Netz zugegriffen werden.

Der beschriebene Vorgang wird rekursiv durchgeführt, wobei sich die Tiefe der Hierarchie nach den Ebenen der verschachtelten Untermodule richtet. Es ist dabei zu beachten, dass sich durch die Neuerstellung eines Moduls für jede Parameterkonfiguration die generierte Hardware "aufbläht". Von *vhisyn* generierte VHDL-Dateien sind dadurch evtl. deutlich unübersichtlicher aufgebaut, als es nötig wäre. Nach dem Evaluierungsschritt in Logiksynthesetools kann dieser Effekt auch mit herkömmlichen Beschreibungsansätzen (Verwendung von **generic** Anweisungen in VHDL) beobachtet werden.

Der Effekt kommt dadurch zustande, dass die Parameter in weiten Teilen die Struktur der Eigenschaften und Kontrollautomaten beeinflussen können, da sie in sämtlichen Ausdrücken auftreten dürfen. Aus diesem Grund wurde auf die Nutzung von VHDL *generics* bei der Übertragung von ITL-Parametern verzichtet und das Modul wird für jede Parameterkonfiguration neu erstellt.

4.3.5 Signalverknüpfungen

Die im **structure** Abschnitt der Vollständigkeitsbeschreibung auftauchenden Verknüpfungen zwischen zwei Signalen können vier verschiedene Arten von Verbindungen herstellen. Diese sind im folgenden aufgelistet:

- Eingang des aktuellen Moduls mit Ausgang des aktuellen Moduls.
- Eingang des aktuellen Moduls mit Eingang eines Submoduls.
- Ausgang eines Submoduls mit Ausgang des aktuellen Moduls
- Ausgang eines Submoduls mit Eingang eines Submoduls.

Dabei müssen jedoch die zwei miteinander verbundenen Signale von ihrem Datentyp zusammenpassen und das Ziel der Verknüpfung darf nicht von mehr als einem Signal getrieben werden, d.h. dass es nicht erlaubt ist, ein Signal dessen Ausgangsverhalten bereits durch Eigenschaften modelliert wurde, in einer Verknüpfung als Ziel zu verwenden. Ebenso dürfen nicht zwei Signale auf das selbe Ziel schreiben.

Wenn ein Ein- oder Ausgang eines Submoduls in der Verknüpfung verwendet wird, ist er durch den Instanznamen zu qualifizieren. Dies geschieht durch Voranstellen des Instanznamens getrennt durch einen Schrägstrich.

4.4 Erstellung des Eigenschaftsgraphen

Der Eigenschaftsgraph $\mathcal{P} = (P, E_{\mathcal{P}})$ besteht aus der Menge der Eigenschaften P, die die Knoten des Graphen darstellen, und einer Menge $E_{\mathcal{P}} \subseteq P \times P$, die

die Kanten des Eigenschaftsgraphen repräsentiert. Der Eigenschaftsgraph ist gerichtet, besitzt keine Mehrfachkanten und kann Zyklen aufweisen.

Im ITL-Quellkode wird zuerst die **reset_property** Anweisung abgearbeitet. Die darin definierte Eigenschaft wird zum noch leeren Graph hinzugefügt und stellt fortan dessen Wurzel dar. Mit

$$reset(\mathcal{P})$$

kann diese Eigenschaft später wieder angesprochen werden.

Die im Abschnitt **property_graph** enthaltenen Anweisungen stellen die Kanten des Graphen dar. Jede Anweisung besteht aus zwei Mengen von Eigenschaften, die je Startknoten (P_{start}) bzw. Zielknoten (P_{ziel}) einer Kante sind. Wie bereits in Abschnitt 3.4.2 angedeutet, ergeben sich die Kanten des Graphen aus dem kartesischen Produkt der Start- und Zielknoten. Dazu wird für jede Anweisung die Menge der Kanten nach folgender Vorschrift aktualisiert:

$$E'_{\mathcal{P}} := E_{\mathcal{P}} \cup (P_{start} \times P_{ziel}) \, .$$

Zusätzlich kann die Menge der Nachfolger einer Eigenschaft $p \in P$ mit

$$succ(p) := \{v \in P \mid (p, v) \in E_{\mathcal{P}}\}$$

definiert werden, ebenso wie sämtliche Vorgänger mit

$$pred(p) := \{v \in P \mid (v, p) \in E_{\mathcal{P}}\}.$$

Die Reseteigenschaft darf dabei als einzige keinen Vorgänger besitzen

$$pred(reset(\mathcal{P})) := \emptyset,$$

während alle Eigenschaften inklusive Reseteigenschaft mindestens einen Nachfolger besitzen müssen, um Deadlocks des Gesamtsystems zu vermeiden

$$\forall p \in P : \; succ(p) \neq \emptyset.$$

4.5 Auswertung der Eigenschaften

Nachdem der Eigenschaftsgraph $\mathcal{P}$ erstellt worden ist, muss jeder Knoten (Eigenschaft) des Graphen einzeln abgearbeitet und getrennt in Hardware umgewandelt werden. Dieser Vorgang soll Inhalt des folgenden Unterkapitels sein. Dazu wird die Eigenschaft ähnlich der Strukturbeschreibung abschnittsweise ausgewertet.

4.5.1 Erfassung der Zeitpunkte

Als erstes werden die Zeitpunkte T im Abschnitt **for timepoints** ausgewertet. Dabei entsteht eine Abbildung

$$timepoint : P \times T \to \mathbb{Z},$$

die dem Namen der Eigenschaft und dem Namen des Zeitpunktes einen konkreten Ganzzahlwert zuordnet. Dazu muss der Ausdruck der Zeitpunktdefinition zu einem konstanten Ausdruck ausgewertet werden.

Die Zeitpunkte beziehen sich in ITL immer auf einen Referenzzeitpunkt t. In dieser Arbeit muss dafür ein konkreter Wert verwendet werden. Dementsprechend wird der Bezeichner t in einem solchen Ausdruck mit dem Wert 0 angenommen. Es könnte jede beliebige Ganzzahl genutzt werden, jedoch erscheint der Wert 0 am intuitivsten.

4.5.2 Erfassung der Grenzen der Eigenschaft

Die beiden Abschnitte **left_hook** und **right_hook** führen ähnlich der Auswertung der Zeitpunkte zu einer Abbildung auf eine Ganzzahl

$$left : P \to \mathbb{Z},$$

$$right : P \to \mathbb{Z}.$$

Es gilt dabei die Bedingung $left(p) < right(p)$.

Da die linke Grenze (**left_hook**) bei der Reseteigenschaft nicht angegeben werden muss, wird sie automatisch zu 0 definiert

$$left(reset(\mathcal{P})) := 0.$$

Zusätzlich wird für den weiteren Verlauf die Länge einer Eigenschaft benötigt. Anhand der Grenzen kann diese als

$$\forall p \in P : \; length(p) := right(p) - left(p)$$

definiert werden.

Eine weitere wichtige Information, die zu diesem Zeitpunkt des Synthesevorgangs gesammelt werden muss, ist der späteste Zeitpunkt der Eigenschaft

$$last : P \to \mathbb{Z},$$

der den Zeitpunkt der letzten Zusicherung oder Annahme darstellt. Er darf jedoch nicht mit der rechten Grenze der Eigenschaft verwechselt werden, da

nach $right(p)$ noch Zusicherungen auftreten dürfen. Dies ist insbesondere bei einer Komponente mit Pipeline-Verhalten der Fall.

Der Zeitpunkt $last(p)$ einer Eigenschaft p kann nicht kleiner sein als $right(p)$. Er wird berechnet, indem alle Zeitpunkte der Annahmen und Zusicherungen (vgl. Abschnitt 3.2.3) zu konstanten Zahlenwerten ausgewertet werden und nur das Maximum dieser Zahlenwerte und $right(p)$ bestimmt wird

$$\forall p \in P : last(p) := max\left(\{right(p)\} \cup \{time(e) \mid e \in (A_p \cup Z_p)\}\right).$$

Da Abschnitt 5.1.2 die Verwendung des **next**-Operators einschränkt, kann vorausgesetzt werden, dass Annahmen und Zusicherungen keine Signalwerte aus der Zukunft zu ihrer Auswertung benötigen. Dadurch ist garantiert, dass $last(p)$ der wirklich letzte Zeitpunkt ist, an dem noch Auswirkungen der betreffenden Eigenschaft vorhanden sind.

4.5.3 Distanz zwischen zwei Eigenschaften

Die Distanz

$$distance : P \times P \to \mathbb{N}$$

zwischen zwei Eigenschaften p_1 und p_2 wird als kürzeste Anzahl Takte definiert, die vergehen müssen, bevor nach dem Beginn einer Instanz von p_1, eine Instanz von p_2 starten kann. Falls im Eigenschaftsgraph $\mathcal{P}$ die Eigenschaft p_2 direkt auf p_1 folgen kann, d.h. es existiert eine Kante $(p_1, p_2) \in E_{\mathcal{P}}$, dann entspricht die Länge $length(p_1)$ der Distanz. Im allgemeinen Fall ist die Distanz jedoch die minimale Summe der Längen der Eigenschaften auf einem Pfad zwischen p_1 und p_2 im Eigenschaftsgraph. Die Länge der Eigenschaft des Zielknotens p_2 wird dabei nicht mit berücksichtigt.

Ein Spezialfall ist die Distanz einer Eigenschaft zu sich selbst. Hierbei wird gefordert, dass der Pfad, der zur Berechnung der kürzesten Distanz herangezogen wird, nicht nur aus einem einzelnen Knoten bestehen darf, d.h. der Pfad besteht aus mindestens zwei Knoten, wobei sein Start- und Endknoten immer die betreffende Eigenschaft selbst ist. Daraus folgt, dass die Distanz zwischen zwei Eigenschaften immer größer oder gleich der Länge der Starteigenschaft ist

$$distance(p_1, p_2) \geq length(p_1).$$

Weiterhin kann die Distanz einer Eigenschaft zu sich selbst auch als Periode dieser Eigenschaft definiert werden

$$period(p) := distance(p, p).$$

Um den Distanzwert zu berechnen, kann ein modifizierter Algorithmus zur Bestimmung des kürzesten Pfades in einem regulären gerichteten und gewichteten Graphen verwendet werden. Dazu bietet sich der Algorithmus nach Dijkstra an, der in vielen Quellen zur Graphentheorie aufgeführt ist, z.B. [Ger99]. Die Modifikation besteht darin, dass als Kantengewicht die Länge der Eigenschaft des Startknotens der Kante genutzt wird. Die Komplexität eines solchen Algorithmus wächst im Allgemeinen linear mit der Anzahl der Eigenschaften und Kanten im Eigenschaftsgraph. Da der Eigenschaftsgraph in der Praxis überschaubar ist, kann die Distanz zwischen zwei Kanten sehr schnell bestimmt werden.

4.5.4 Freezevariablen

Jede Definition einer Freezevariable $f \in F$ besteht aus drei Teilen: dem Namen der Variable, einem beliebigen Ausdruck und dem Speicherzeitpunkt, zu dem der Ausdruck in die Freezevariable übertragen wird. Im ersten Schritt wird der Ausdruck auf einen Hardwareblock abgebildet, der zu jedem beliebigen Zeitpunkt den Wert des Freezevariablenausdrucks bereitstellt.

Zur formalen Definition werden an dieser Stelle zwei Funktionen eingeführt, die die entsprechende Freezevariable auf den Wert des Freezeausdrucks bzw. den Speicherzeitpunkt der Variable abbilden

$$expr : F \rightarrow \textit{Netz-ID},$$

$$time : F \rightarrow \mathbb{Z}.$$

Das Intervall, in dem der Wert der Freezevariable f bereitgestellt werden muss, beginnt mit dem Speicherzeitpunkt $time(f)$ und endet mit dem letzten Zeitpunkt $last(p)$ der Eigenschaft p. Damit die Freezevariable im gesamten Interval den korrekten Wert hat, und nicht nur zum Speicherzeitpunkt $time(f)$, muss der Wert von $expr(f)$ zwischengespeichert werden.

Im Folgenden werden zwei Implementierungsvarianten zur Speicherung der Freezevariablen vorgestellt und bezüglich ihres voraussichtlichen Hardwareaufwandes untersucht. Einige Überlegungen dazu finden sich in der Diplomarbeit von Pepelyashev [Pep09], die zu der hier vorgestellten zweiten Implementierungsvariante führten.

Registerkette

Damit der Wert des Freezeausdrucks nach dem Speicherzeitpunkt noch verwendet werden kann, muss er mit Hilfe von Registern zwischengespeichert werden. Eine erste und einfache Umsetzung sah vor, dass für jeden betroffenen Zeitpunkt ein eigenes Register zur Verfügung stand. Damit wird bei sehr

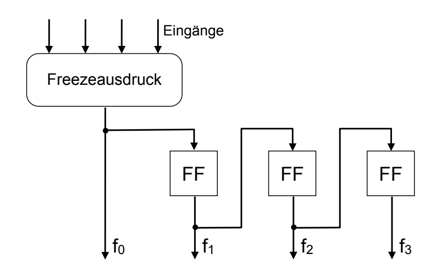


Abbildung 4.3: Zwischenspeichern einer Freezevariable mit Hilfe einer Kette von Registern.

langen Eigenschaften eine große Menge an Registern benötigt. In Abb. 4.3 ist die dabei entstehende Hardware abgebildet. Der Freezeausdruck legt dabei den zu speichernden Wert in jedem Zeitschritt an f_0 an, wobei er natürlich nur zum Zeitpunkt $time(f)$ auch gültig ist. An den Ausgängen f_i liegt somit immer zum Zeitpunkt $time(f)+i$ der korrekte Wert an. Die Kette muss dabei so lang sein, dass für das gesamte Freezeinterval ein Ausgang zur Verfügung steht. Der maximale Wert von i ist

$$last(p) - time(f).$$

Analyse der Eigenschaftsüberlappung

Durch den beschriebenen Ansatz, Freezevariablen abzubilden, lassen sich sehr lange, gering überlappende Eigenschaften nur sehr ineffizient synthetisieren. Das ist vor allem damit begründet, dass zum einen viele Register notwendig sind, zum anderen aber auch die Optimierung der Zusicherungen nicht möglich ist (siehe Abschnitt 4.5.6). Aus diesem Grund ist es notwendig, eine Freezevariable in einem Register zu speichern und dann in diesem so lange wie möglich zu halten.

Zur Bestimmung der Haltedauer muss das Überlappen der Eigenschaften (z.B. für Pipelineverhalten) berücksichtigt werden. Dies ist notwendig, da eine Eigenschaft, die sich selbst überlappt, den Freezevariablenspeicherplatz der Vorgängereigenschaft (d.h. die vorhergehende Instanz der selben Eigenschaft) überschreibt. Mit Hilfe der Implementierung durch eine Kette von Registern ist das korrekte Verhalten bei Überlappung automatisch erreicht, allerdings auf Kosten der Hardwareressourcen.

Die Zahl der Register für Freezevariablen kann bei langen Eigenschaften um ein vielfaches verringert werden, indem die Registerwerte in der Kette

nur dann weitergeschoben werden, wenn eine Überlappung droht, d.h. wenn eine zweite Instanz der selben Eigenschaft aktiviert wird, während die erste Instanz noch nicht komplett abgearbeitet ist ($last(p)$ erreicht ist). Der Zeitraum zwischen der Bereitstellung des Freezeausdrucks und der drohenden Überlappung entspricht der Distanz der Eigenschaft zu sich selbst, d.h. der Periode der Eigenschaft.

Falls die Berechnung der Periode der Eigenschaft zu aufwendig ist, kann die Länge $length(p)$ der Eigenschaft als untere Schranke genutzt werden.

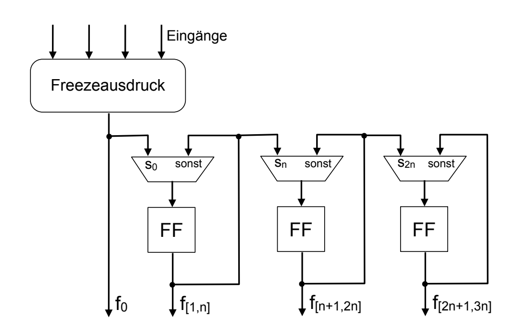


Abbildung 4.4: Zwischenspeichern einer Freezevariablen anhand der Periode n der Eigenschaft.

In Abb. 4.4 ist die nötige Hardware für ein solches Verhalten dargestellt. Vor jedes Register wird ein Multiplexer eingebaut, der entweder den Wert des Registers hält oder den der vorhergehenden Stufe übernimmt. Das Schaltsignal für die Übernahme des neuen Wertes wird aus dem Zustandsautomaten der Gesamtkomponente bestimmt. Das Signal s_i wird genau dann zugeschalten, wenn der Zustandsautomat eine Instanz der betreffenden Eigenschaft p aktiviert hat und diese sich im Zeitpunkt $time(f) + i$ befindet. In Abschnitt 4.8 wird die Erzeugung des Schaltsignals erläutert und dafür die Notation $\psi_{p,t}$ verwendet.

Der Freezeausdruck muss direkt zum Speicherzeitpunkt $time(f)$ (s_0) und danach alle n Takte ins nächste Register weitergeschoben werden. Der Wert von n ist dabei die Periode der Eigenschaft $period(p)$.

Die Ausgänge der abgebildeten Hardware sind mit einem Intervall gekennzeichnet, welches die Zeitpunkte umfasst, zu dem der Wert der Freezevariable f von diesem Signal abgegriffen werden kann. Direkt zum Zeitpunkt $time(f)$ muss der Wert wie schon bei der Registerkettenimplementierung am Ausgang der Hardware des Freezeausdrucks genutzt werden.

Die Anzahl der Stufen und damit Register in dieser Implementierungsvariante ergibt sich aus

$$\left\lceil \frac{last(p) - time(f)}{period(p)} \right\rceil .$$

Insbesondere bei sehr langen Eigenschaften bleibt dieser Wert klein, da Zähler und Nenner gleichermaßen wachsen. Bei hoher Überlappung dagegen, kann dieser Wert schnell wachsen, und man hat keine Vorteile gegenüber der Registerkettenimplementierung.

Illustration des Registerschedulings an einem Beispiel

Zur Verdeutlichung des Verhaltens der beiden vorgestellten Ansätze soll Abb. 4.5 dienen. Sie zeigt eine beliebige Eigenschaft p mit einer Freezevariablen f. Die Eigenschaft beginnt mit $left(p) = 0$ und endet zu $right(p) = 2$. Die Grenzen sind in der Abbildung durch kleine Dreiecke symbolisiert. Aus den Grenzen ergibt sich eine Länge $length(p) = 2$. Der letzte Zeitpunkt $last(p) = 4$ der Eigenschaft verursacht ein überlappendes Verhalten. Weiterhin muss sichergestellt sein, dass der Eigenschaftsgraph $\mathcal{P}$ eine Kante $(p, p) \in E$ enthält, sodass die Eigenschaft auf sich selbst folgen und sich damit selbst überlappen kann.

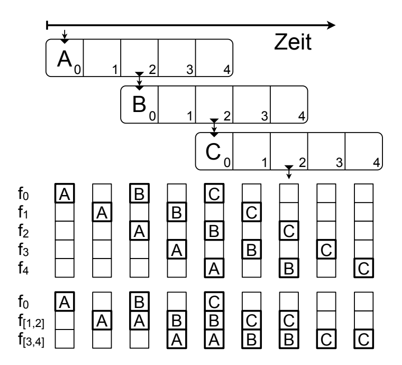


Abbildung 4.5: Speichern einer Freezevariable bei überlappenden Eigenschaften.

Im dargestellten Beispiel sind drei Instanzen dieser Eigenschaft hintereinander aktiviert worden. Es ist erkennbar, dass die Eigenschaften sich mit jeweils drei ihrer fünf Zeitpunkte überlappen. Der letzte Zeitpunkt $t+4$ kann sogar zu einer dreifachen Überlappung führen. Wenn man die Kette fortsetzt,

sind allerdings keine höheren Überlappungen möglich, weswegen das Beispiel nur drei Instanzen umfasst.

Der Speicherzeitpunkt $time(f) = 0$ liegt zu Beginn der Eigenschaft und bewirkt, dass der Freezeausdruck jeweils mit dem Start jeder Instanz am Ausgang f_0 der Hardware des Freezeausdrucks anliegt. Demnach wird für f_0 kein Register benötigt. Die drei in diesem Beispiel anliegenden Werte des Freezeausdrucks zu den drei verschiedenen Speicherzeitpunkten werden mit A, B und C benannt.

Weiterhin sind im unteren Teil der Abbildung die benötigten Register für beide Implementierungen zu sehen. Die oberen Register sind mit f_1 bis f_4 bezeichnet und stellen die Register der Kettenimplementierung dar. Dabei wird mit jedem Zeitschritt der Wert des Freezeausdrucks von einem in das nächste Register übernommen. Es werden insgesamt vier Register benötigt, da zwischen Speicherzeitpunkt $time(f)$ und letztem Zeitpunkt der Eigenschaft $last(p)$ vier Takte vergehen. Wenn ein Register zu einem Zeitpunkt einen gültigen Freezeausdruck speichert, ist das Register in der Abbildung entsprechend beschriftet und hervorgehoben. Da nur alle zwei Takte eine neue Instanz anfängt aber jedoch in jedem Takt der Wert weitergeschoben wird, ist die Auslastung der Register mit gültigen Werten entsprechend gering. Bei höheren Überlappungen wird dieser Effekt immer stärker.

Die untere Reihe Register stellt die Implementierung dar, die die Länge der Eigenschaft in Betracht zieht. Dabei wird der Wert nur alle zwei Takte ins nächste Register geschoben. Dadurch sind die beiden benötigten Register viel besser ausgelastet als in der ersten Variante. Der Ausgang $f_{[1,2]}$ stellt somit den Wert der Freezevariable jeweils zu den Zeitpunkten 1 und 2 bereit, während der Ausgang $f_{[3,4]}$ den Wert zu den verbleibenden beiden Zeitpunkten bereitstellt.

Gegenüberstellung der Implementierungsvarianten

Beide Implementierungen haben Vor- und Nachteile. Die Kette von Registern kommt mit nur einem Register pro Stufe aus, während die Periodenimplementierung einen zusätzlichen Multiplexer verwenden muss. Auf der anderen Seite kann bei großen Perioden und langen Eigenschaften die Anzahl der Stufen mit der zweiten Variante deutlich reduziert werden. In allen anderen Fällen sollte auf die Registerkettenimplementierung zurückgegriffen werden.

Bei beiden Varianten muss natürlich beachtet werden, dass die Register und eventuellen Multiplexer in ihrer Breite nicht nur auf ein Bit ausgelegt sein dürfen, sondern die volle Bitbreite des Freezeausdrucks unterstützen müssen.

Speichern der Freezevariablen im lokalen Signalverzeichnis

Beginnend mit dem Speicherzeitpunkt $time(f)$ kann der Bezeichner der Freezevariable f in den Ausdrücken innerhalb der Eigenschaft verwendet werden. Seine Sichtbarkeit beschränkt sich dabei auf weitere Freezevariablendefinitionen derselben Eigenschaft, sowie deren Annahmen und Zusicherungen. Innerhalb von Makros sind Freezevariablen nicht sichtbar.

Der Zeitpunkt, zu dem f benutzt wird, hängt nicht nur von dem jeweiligen Zeitpunkt hinter dem **at** Schlüsselwort ab, sondern kann sich durch entsprechende Verwendung von **prev** und **next** Operatoren wieder verschieben.

Durch die Implementierungen, die bereits vorgestellt wurden, stehen die Netz-ID und Signalbelegung (falls der Freezeausdruck zu einer Konstante wird, sogar der konkrete Wert) zu jedem Zeitpunkt i im Freezeintervall bereit und kann entsprechend im lokalen Signalverzeichnis der Eigenschaft abgelegt werden

$$\forall i \in [time(f), last(p)] : locals(f, i) := f_{i-time(f)}.$$

Bei der Auswertung eines Ausdrucks wird der Auswertungszeitpunkt mitgeführt. Wenn dabei der Bezeichner einer Freezevariable auftaucht, kann er mit Hilfe dieses Zeitpunktes im lokalen Signalverzeichnis nachgeschlagen werden. In Abschnitt 4.9.1 wird dieser Vorgang aus Sicht der Hardwareabbildung beschrieben.

4.5.5 Auswertung der Annahmen

Die Menge der Annahmen einer einzelnen Eigenschaft $p \in P$ wird mit A_p bezeichnet während die Menge aller Annahmen durch

$$A := \{A_p \mid p \in P\}$$

dargestellt wird.

Eine Annahme $a \in A$ besteht aus einem Ausdruck, der ähnlich den Freezevariablen direkt in eine Hardwarerepräsentation überführt werden kann und einem Zeitpunkt, zu dem der Ausdruck ausgewertet werden soll. Der Ausdruck kann mit

$$expr : A \rightarrow \textit{Netz-ID}$$

nachgeschlagen werden, wobei *Netz-ID* das Ausgangsnetz der entsprechenden Hardwarerepräsentation ist. Weiterhin wird mit

$$time : A \rightarrow \mathbb{Z}$$

der Zeitpunkt der Annahme angesprochen. Die beiden Funktionen stimmen mit den korrespondierenden Funktionen der Freezevariablen überein. Anhand

des übergebenen Funktionsargumentes ist die Zuordnung allerdings ersichtlich.

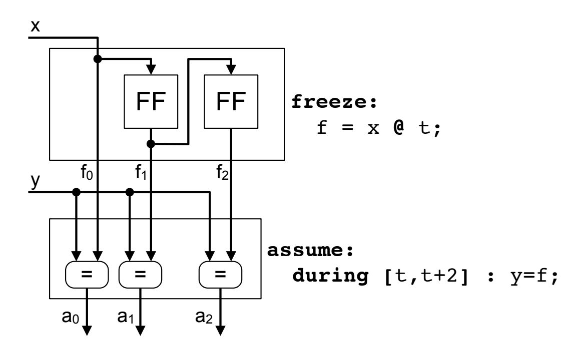


Abbildung 4.6: Beispiel für unterschiedliche Hardware einer Annahme, die mit **during** zu drei Zeitpunkten aktiv ist.

An dieser Stelle soll angenommen werden, dass Annahmen in einer **during** Anweisung bereits aufgeteilt wurden, sodass für jeden Zeitpunkt im **during** Intervall eine einzelne Annahme vorliegt. Obwohl der Freezeausdruck gleich ist, kann die entstehende Hardware zu jeder dieser einzelnen Annahmen nicht zusammengefasst werden. Zur Illustration dieses Umstandes soll das Beispiels in Abb. 4.6 dienen. Eine Freezedefinition f speichert den Eingang x zum Zeitpunkt 0 (t wird zu 0 angenommen) in der Variable f_0 und schiebt ihn mit einer Registerkette an die Signale f_1 und f_2. Der jeweilige Wert wird in einer Annahme während der Zeitpunkte 0 bis 2 mit dem Eingang y verglichen. Dabei muss der Vergleichsoperator (die Hardware des Ausdrucks der Annahme) drei mal instanziiert werden, da jede Instanz mit einem anderen Signal für die Freezevariable rechnen muss. Man sieht, dass die drei Vergleichsoperatoren nicht ohne weiteres zusammengefasst werden können.

Da nicht jede Annahme unbedingt in einem logischen Wahrheitswert resultiert und als Ergebnis auch einen Mehrbitvektor liefern kann, muss dieser Vektor in einem Zwischenschritt in einen logischen Wahrheitswert (Einbitvektor) umgewandelt werden. Dies geschieht, indem das Ergebnis auf Gleichheit mit dem Nullvektor geprüft wird. Der Vergleich kann auch mit Hilfe des logischen disjunktiven Reduktionsoperators (in Verilog *reduction or*) durchgeführt werden. An einem Beispiel betrachtet, geschieht die Umwandlung der Annahme

```
at t: vec1 + 5;
```

in die funktional äquivalente Annahme

```
at t: (vec1 + 5) /= 0;
```

mit nur einem Bit Breite.

Man erhält nach diesem Schritt eine Menge einzelner Annahmen. Dabei kann es auch vorkommen, dass mehrere Annahmen zum gleichen Zeitpunkt auftreten. Diese werden dann zu einer Gesamtannahme

$$\hat{a}_{p,i} := \bigwedge^{a \in A_{p,i}} a$$

mit

$$A_{p,i} := \{a \in A_p \mid time(a) = i\}$$

zusammengefasst, wobei die Menge $A_{p,i}$ alle Annahmen einer Eigenschaft p zum Zeitpunkt i beinhaltet. Die Elemente von $A_{p,i}$, bzw. deren Annahmenausdrücke, werden danach konjunktiv verknüpft und bilden somit die Gesamtannahme für diesen Zeitpunkt. Das resultierende Signal der Gesamtannahme kann später bei der Synthese des Eigenschaftsgraphen direkt verwendet werden. Falls die Menge $A_{p,i}$ der Annahmen zu einem bestimmten Zeitpunkt leer ist, wird die Gesamtannahme zur so genannten *Trivialannahme*, die immer zu 1 ausgewertet wird

$$A_{p,i} = \emptyset \;\Rightarrow\; \hat{a}_{p,i} := 1.$$

Die Menge der Gesamtannahmen $\hat{A}$ kann mit

$$\hat{A} := \{\hat{a}_{p,i} \mid p \in P, i \in \mathbb{Z}, A_{p,i} \neq \emptyset\}$$

zusammengefasst werden und schliesst alle Trivialannahmen aus.

Weiterhin muss sichergestellt werden, dass keine Einzelannahmen vor der linken Grenze der Eigenschaft liegen. Dies kann durch einfaches Verzögern des Annahmenausdrucks um die entsprechende Anzahl Takte gewährleistet werden. In unserer mathematischen Darstellung wird davon ausgegangen, dass dieser Schritt bereits durchgeführt wurde und die Bedingung

$$\forall a \in A_p : time(a) \geq left(p)$$

für jede Eigenschaft $p \in P$ gilt.

4.5.6 Auswertung der Zusicherungen

Bei den Zusicherungen reicht es im Gegensatz zu den Annahmen nicht den gesamten Ausdruck der Zusicherung direkt in Hardware abzubilden. Da sich aus ihr die Zuweisung eines Wertes an die Ausgänge ableiten muss, wird gefordert, dass jede Zusicherung in Form einer Signalzuweisung auftritt. In Abschnitt 3.2.3 ist die dafür erforderliche Syntax in ITL dargestellt. Die Teile einer Zusicherung werden ähnlich den vorangegangenen Definitionen angesprochen, wobei lediglich die Abbildung *target* neu hinzukommt und das Zuweisungsziel der Zusicherung in Form eines Signalnamens angibt

$$target : Z \rightarrow Name,$$

$$expr : Z \rightarrow Netz\text{-}ID,$$

$$time : Z \rightarrow \mathbb{Z}.$$

Zur Vereinfachung der Handhabbarkeit der Zusicherungen wird eine zusätzliche Notation eingeführt, die es erlaubt, zu einem gegebenen Tupel aus Eigenschaft p, Signalname n und Zeitpunkt i die passende Zusicherung z zu finden

$$\begin{aligned} z_{p,n,i}: \quad & P \times Name \times \mathbb{Z} \rightarrow Netz\text{-}ID, \\ & (p, n, i) \mapsto z \in Z_p \mid target(z) = n \wedge time(z) = i, \end{aligned}$$

wobei Z_p die Menge aller Zusicherungen einer bestimmten Eigenschaft $p \in P$ ist. Es wird davon ausgegangen, dass nur exakt eine Zusicherung zu jeder Kombination von Eigenschaftszeitpunkt und Name auftaucht, d.h. die Eindeutigkeit der Abbildung muss gewährleistet sein. Wenn ein solcher Fall auftritt, erkennt *vhisyn* die mehrfache Zuweisung und meldet dies dem Benutzer.

Die Gesamtmenge der Zusicherungen kann damit wie folgt dargestellt werden

$$Z := \bigcup_{p}^{P} Z_p.$$

Auffinden äquivalenter Zusicherungen

An dieser Stelle könnte eigentlich die Auswertung der Einzeleigenschaften abgeschlossen werden. Im Falle von sehr langen Eigenschaften und Zusicherungen in Form von langen **during** Anweisungen darf die Hardwareabbildung nicht auf die gleiche Art und Weise geschehen, wie die der Annahmen. Der Unterschied zwischen Annahmen und Zusicherungen ist darin begründet,

dass Annahmen meist nur kurze Aktivierungsbedingungen beinhalten, während Zusicherungen teilweise komplexe Berechnungsergebnisse über längere Zeiträume an Ausgangssignale zuweisen. Wenn die benötigte Hardware über das gesamte **during** Intervall einzeln erzeugt wird, entsteht eine große, "aufgeblähte" Repräsentation, die dann erst vom Logiksynthesewerkzeug wieder optimiert werden kann. Aus diesem Grund muss untersucht werden, inwieweit äquivalente Zusicherungen schon vor der Erzeugung der Hardware zusammengefasst werden können.

Die gewählte Implementierungsstrategie sieht eine Unterteilung isomorpher Zusicherungsausdrücke in Intervalle vor. Für jedes Intervall wird dann der Ausdruck nur einmal ausgewertet, bzw. es wird nur eine einzige Hardwareinstanz benötigt. In einem ersten Schritt ist es dazu notwendig herauszufinden, welche Freezevariablen zu welchen Zeitpunkten von einer Zusicherung $z \in Z$ verwendet werden. Die Menge der Variablen und Zeitpunkte kann durch die Abbildung

$$fsupp : Z \to F \times \mathbb{Z}$$

bestimmt werden, indem der abstrakte Syntaxbaum (AST) des Ausdrucks von der Wurzel beginnend durchsucht (traversiert) wird und die Vorkommen von Freezevariablen gespeichert werden. Der Zeitpunkt wird dabei initial auf den Wert $time(z)$ gesetzt und rekursiv während der Traversierung aktualisiert. Jedes Vorkommen des **prev** Operators setzt den Zeitwert um den Wert des zweiten Arguments (default: 1) zurück, während der **next** Operator eine Erhöhung verursacht. Makroaufrufe müssen dabei auch berücksichtigt werden, obwohl in ihnen keine Freezevariablen sichtbar sind, denn etwaige **prev** oder **next** Operatoren auf tieferen Ebenen können eine Verschiebung des Benutzungszeitpunktes in den Argumenten des Makros hervorrufen.

Ein Beispiel kann dies verdeutlichen. Die Zusicherung mit dem ITL-Quelltext

```
at t+3 : sig = prev(f) + makro1(f);
```

würde im Falle einer Implementierung des Makros *makro1* als

```
macro makro1(x : numeric) : numeric :=
  prev(f,2) + f;
end macro;
```

zu einer Menge

$$fsupp(z) := \{(f, 1), (f, 2), (f, 3)\}$$

führen, da die Freezevariable f, jeweils zu den Zeitpunkten $t + 1$, $t + 2$ und $t + 3$ abgegriffen wird.

In einem zweiten Schritt werden die Tupel in der $fsupp$ Menge in Äquivalenzklassen eingeteilt, die den Stufen der Freezevariablenimplementierung entsprechen. Wenn in obigem Beispiel die Freezevariable f zum Zeitpunkt $t+2$ und $t+3$ aus dem Ausgang der selben Registerstufe (bspw. $f_{[1,2]}$) stammt und zum Zeitpunkt $t+1$ unmittelbar aus dem Netz des Freezeausdrucks f_0 kommt, kann die Menge $\{(f,1),(f,2),(f,3)\}$ in

$$fsupp(z)/\!\sim\; := \{f_0, f_{[1,2]}\}$$

überführt werden. Man beachte an dieser Stelle, dass $time(f) := 1$ angenommen wurde, sodass die Indices der zweiten Menge um 1 nach vorn verschoben sind.

Der dritte und letzte Schritt bestimmt alle Zusicherungen, deren Ausdrücke (bzw. die entsprechenden Hardwarerepräsentationen) isomorph sind ($z_1 \cong z_2$) und deren $fsupp(z)/\!\sim$ gleich ist. Für diese "gleichen" Zusicherungen wird dann nur eine Instanz der nötigen Hardware implementiert. In der folgenden Bedingung gibt der Ausdruck $z_1 \sim z_2$ an, dass zwei Zusicherungen äquivalent ("gleich") sind

$$z_1 \sim z_2 \quad :\Longleftrightarrow \quad z_1 \cong z_2 \wedge fsupp(z_1)/\!\sim\; = fsupp(z_2)/\!\sim$$

Basierend auf dieser Gleichheit kann nun eine Menge der Äquivalenzklassen aller Zusicherungen definiert werden

$$Z/\!\sim\; := \{[z] | z \in Z\},$$

deren Elemente alle gleichen Zusicherungen enthalten. Diese Elemente $[z]$ sind wie folgt bestimmt

$$[z] := \{z_i \in Z \mid z_i \sim z\}\,.$$

Für jedes Element $[z]$ wird während der Generierung der Hardware nur eine Hardwarerepräsentation erzeugt. Jede Zusicherung $z_i \in [z]$ nutzt demnach das selbe Ausgangssignal dieser Hardware. Die vorliegenden Definitionen werden in Abschnitt 4.9.4 genutzt, um die Zuweisung der Zusicherungen zu vereinfachen.

Wenn die Freezevariablen mit Hilfe der Registerkettenvariante implementiert werden, birgt diese Art der Zusammenfassung von Zusicherungshardware kaum Vorteile. Lediglich bei Ausdrücken, die ohne Freezevariablen auskommen, kommt es hier zu Einsparungen. Wenn allerdings die Eigenschaften sehr lang werden und umfangreiche **during** Blöcke die Zusicherungen dominieren, kann es zu massiven Verbesserungen des nötigen Ressourcenaufwandes kommen.

Theoretisch könnte das Auffinden dieser Einsparungen auch dem im Entwurfsfluss nachgeordneten Logiksynthesewerkzeug überlassen werden. Da aber allein schon die Laufzeit von *vhisyn* bzw. der Umfang der generierten Hardware zu groß wird, ist das Finden äquivalenter Zusicherungen in der Praxis für viele Beispiele ein erforderlicher Schritt, um handhabbare Hardware zu erzeugen.

4.6 Konstruktion des nichtdeterministischen Kontrollautomaten

Der nächste Schritt im Syntheseprozess von Operationseigenschaften ist die Konstruktion des Kontrollautomaten der Komponente, der später in einem weiteren Schritt in Hardware abgebildet werden muss.

In dieser Arbeit wird der Automat direkt aus dem Eigenschaftsgraphen hergeleitet [LH09]. Die Vorteile liegen in der meist recht geringen Komplexität des Eigenschaftsgraphen, weshalb der entstehende Automat sehr schnell erstellt werden kann und eine überschaubare Komplexität besitzt. Um den Konstruktionsvorgang besser verstehen zu können, soll zuerst der Eigenschaftsgraph genauer untersucht werden.

Zu Beginn muss der Begriff der Eigenschaftsposition definiert werden. Eine Eigenschaftsposition besteht aus einer Eigenschaft p, sowie einem Zeitpunkt t innerhalb dieser Eigenschaft und wird durch p^t dargestellt. Die Menge der Eigenschaftspositionen aller Eigenschaften wird mit

$$Q := \{p^t \mid p \in P, t \in \mathbb{Z}, left(p) \leq t \leq last(p)\}$$

bezeichnet.

Analog zur Reseteigenschaft des Eigenschaftsgraphen kann eine Reseteigenschaftsposition definiert werden, die sich aus der linken Grenze der Reseteigenschaft ergibt

$$\begin{aligned} Q_0 &:= reset(\mathcal{P})^{left(reset(\mathcal{P}))} \\ &:= reset(\mathcal{P})^0. \end{aligned}$$

4.6.1 Abgerollter Eigenschaftsgraph

Mit Hilfe der im Abschnitt 4.5.2 definierten Eigenschaftsgrenzen kann ein abgerollter Eigenschaftsgraph $\mathcal{U} = (Q, E_{\mathcal{U}})$ erstellt werden. Die Knoten dieses Graphen bestehen aus den Zeitpunkten aller beteiligten Eigenschaften P, den Eigenschaftspositionen.

Die Kanten $E_{\mathcal{U}}$ des abgerollten Eigenschaftsgraphen sind in zwei Kategorien zu unterscheiden. Zum einen gibt es die Kanten, die den temporalen Verlauf einer Eigenschaft sicherstellen. Diese Übergänge werden Zeitkanten E_{temp} genannt

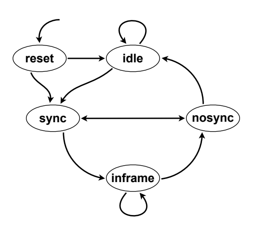
$$E_{temp} := \{(p^t, p^{t+1}) \mid p \in P \wedge left(p) \leq t < last(p)\}$$

und repräsentieren das Fortschreiten der Eigenschaften mit jedem Zeitschritt. Die zweite Kategorie bilden die Kanten, die einen Übergang von einer Eigenschaft zu einem ihrer Nachfolger darstellen. Sie werden Nachfolgerkanten E_{succ} genannt

$$E_{succ} := \{(p_1^{right(p_1)}, p_2^{left(p_2)}) \mid p_1 \in P \wedge p_2 \in succ(p_1)\}.$$

Somit ergibt sich die Menge $E_{\mathcal{U}}$ aller Kanten des neuen Graphen aus der Vereinigung dieser beiden Teilmengen

$$E_{\mathcal{U}} := E_{temp} \cup E_{succ}.$$



(a) Eigenschaftsgraph

Eigenschaft	$left(p)$	$right(p)$	$last(p)$
reset	0	3	4
idle	0	1	2
sync	2	8	9
inframe	0	8	9
nosync	0	3	4

(b) Grenzen der Eigenschaften

Abbildung 4.7: Komponente *Framer* mit Eigenschaftsgraph und -grenzen.

Der abgerollte Eigenschaftsgraph lässt sich am besten mit Hilfe eines Beispiels verstehen. Da das bisher genutzte *Resampler* Beispiel nicht komplex genug ist, um einige der Attribute des Graphen klar darzustellen, soll ein

neues Beispiel mit Namen *Framer* eingeführt werden. In Abb. 4.7 sind dazu der Eigenschaftsgraph sowie die Grenzen der Eigenschaften abgebildet. Der weitere Aufbau der Eigenschaften ist an dieser Stelle nicht notwendig. Der dazu gehörige abgerollte Eigenschaftsgraph ist in Abb. 4.8 zu sehen. Jedes nummerierte Kästchen entspricht darin einer Eigenschaftsposition.

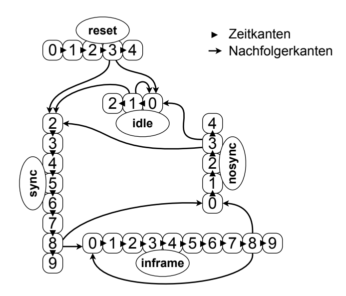


Abbildung 4.8: Abgerollter Eigenschaftsgraph der Komponente *Framer*.

Eine weitere Besonderheit ist der Beginn einer Eigenschaft. Die Syntax von ITL erlaubt Annahmen und Zusicherungen vor der linken Grenze. Die in dieser Arbeit vorgestellten Algorithmen zur Konstruktion eines Kontrollautomaten erfordern jedoch, dass Annahmen und Zusicherungen erst mit der linken Grenze beginnen dürfen. Im Falle von Annahmen kann dies durch eine Verzögerung der Annahme auf den Zeitpunkt der linken Grenze erreicht werden. Bei Zusicherungen ist dieses Verfahren nicht möglich, da die Zuweisung an bspw. ein Ausgangssignal nicht verzögert werden darf und sofort geschehen muss. Aus diesem Grund sind zu frühe Zusicherungen nicht erlaubt. In Abschnitt 5.1.4 wird dies näher erläutert.

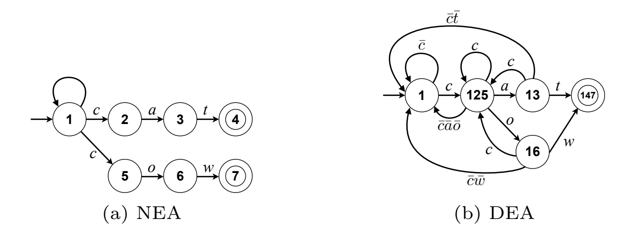
Bei der Betrachtung des abgerollten Eigenschaftsgraphen ist zu beachten, dass dieser keinen normalen Automaten darstellt, jedoch wie ein Automat abgearbeitet werden kann. Der dieser Abarbeitung zugrunde liegende Automat stellt das Ergebnis des im folgenden vorgestellten nichtdeterministischen Konstruktionsverfahrens dar.

4.6.2 Nichtdeterministische endliche Automaten

Zuvor muss allerdings geklärt werden, was ein NEA ist. Entsprechend [HMU06] unterscheidet sich ein nichtdeterministischer von einem deterministischen endlichen Automaten (DEA) durch Zustandsübergänge, deren Übergangsbedingungen sich nicht gegenseitig ausschliessen. Damit können zu einem bestimmten Zeitpunkt mehrere mögliche Nachfolgezustände aktiviert

werden. Synthetisierbare Hardwarebeschreibungen können prinzipiell nur deterministisches Verhalten aufweisen. Wenn ein NEA in Hardware abgebildet werden soll, muss dieser vorher entweder in einen äquivalenten DEA umgewandelt oder sein Verhalten durch einen DEA "emuliert" werden, wie dies in [SP01] beschrieben wird. Auf die dabei entstehenden Probleme wird im weiteren Verlauf dieses Abschnitts eingegangen.

Um die Notwendigkeit von nichtdeterministischen Automaten in verschiedenen Anwendungsbereichen zu demonstrieren, soll ein Beispiel präsentiert werden. Beim automatisierten Verarbeiten von Zeichenketten kommen oft reguläre Ausdrücke (engl. Regular Expressions (REs)) zum Einsatz (vgl. Abschnitt 2.2.1). Diese Ausdrücke entsprechen NEAs, deren Eingabe die zu verarbeitende Zeichenkette ist und deren Ausgabe eine Ja/Nein-Entscheidung darstellt, die angibt, ob der reguläre Ausdruck auf die Zeichenkette passt (*match*).



(a) NEA (b) DEA

Abbildung 4.9: Beispiel eines endlichen Automaten zur alternativen Erkennung der Zeichenketten "cat" bzw. "cow" in einem Text.

Zur Illustration wird auf eine beliebige Zeichenkette des ASCII-Zeichensatzes der reguläre Ausdruck ($cat|cow$) angewendet, d.h. wenn in der Zeichenkette eine Teilzeichenkette "cat" oder "cow" auftritt, soll der Automat dies als Erfolg signalisieren. In Abb. 4.9 ist der dazugehörige NEA dargestellt. Weiterhin zeigt die Abbildung zum Vergleich einen entsprechenden DEA, der die selbe Funktion erfüllt, jedoch deutlich mehr Übergänge besitzt und damit wesentlich unübersichtlicher ist. Die jeweiligen Endzustände sind durch einen inneren Kreis gekennzeichnet und der Automat signalisiert bei Erreichen eines dieser Zustände das erfolgreiche Erkennen des regulären Ausdrucks.

4.6.3 Formales Konstruktionsverfahren

NEA als 6-Tupel

Der für einen gegebenen Eigenschaftsgraphen zu konstruierende NEA wird durch ein 6-Tupel

$$\mathcal{A}_{NEA} = (S, S_0, \Sigma, \Omega, \delta, \lambda)$$

charakterisiert, bei dem

- S die Menge der Zustände,
- S_0 der Startzustand $S_0 \in S$,
- Σ das Eingabealphabet,
- Ω das Ausgabealphabet,
- δ die Zustandsübergangsrelation $\delta \subseteq S \times \Sigma \times S$,
- sowie λ die Ausgaberelation $\lambda \subseteq S \times \Omega$

sind. Damit ist der Automat ein Moore-Automat, bei dem die Ausgaberelation nicht vom Eingabealphabet abhängt.

Zustände

Die Menge der Zustände

$$S := 2^Q$$

besteht aus der Potenzmenge der Eigenschaftspositionen Q. Dabei enthält jeder Zustand in S eine oder mehrere Eigenschaftspositionen aus Q. Der Startzustand S_0 besteht aus der Reseteigenschaftsposition Q_0

$$S_0 := \{Q_0\},$$

die, wie bereits definiert, den Anfang der Reseteigenschaft darstellt. Es ist zu beachten, dass S_0 nicht exakt Q_0 entspricht, sondern einer Menge mit einen einzigen Element Q_0.

Eingabealphabet

Das Eingabealphabet Σ beinhaltet wiederum die Potenzmenge aller in Abschnitt 4.5.5 definierten Gesamtannahmen $\hat{A}$

$$\Sigma := 2^{\hat{A}}.$$

Zustandsübergangsrelation

Die Zustandsübergangsrelation $\delta \subseteq S \times \Sigma \times S$ beinhaltet alle Transitionen des Automaten. Dabei werden ausgehend von einem bestimmten Zustand $s \in S$ seine Übergangsbedingungen $\sigma_s \in \Sigma$ und Zielzustände $succ(s)$ definiert, und die entstehenden Tupel zu δ hinzugefügt

$$\delta := \{(s, \sigma, s') \mid s \in S, s' \in succ(s), \sigma_s \subseteq \sigma\}.$$

Die dabei notwendige Definition von

$$\sigma_s := \{\hat{a}_{p,t} \mid \hat{a}_{p,t} \in \hat{A}, p^t \in s\}$$

sammelt alle Gesamtannahmen des Ausgangszustands. Dabei muss jede einzelne der Annahmen erfüllt sein, damit der Zustandsübergang durchgeführt werden kann. Es ist natürlich auch erlaubt, dass mehr Annahmen erfüllt sind als in σ_s enthalten sind.

Der nächste Konstruktionsschritt ist das Finden der Nachfolger für jeden Zustand, d.h. die Funktion $succ(s)$ muss definiert werden. Dazu muss zuerst die Nachfolgermenge einer einzelnen Eigenschaftsposition berechnet werden. Im Normalfall wird einfach der Eigenschaftszeitpunkt um eins erhöht. Das entspricht den Zeitkanten im abgerollten Eigenschaftsgraph.

Spezielle Beachtung findet der Fall, dass sich die Eigenschaftsposition unmittelbar vor ihrer rechten Grenze befindet ($t + 1 = right(p)$), weil dann die linken Grenzen aller Nachfolgereigenschaften mit in den Zustand aufgenommen werden müssen. Wenn der Zeitpunkt nicht um eins erhöht werden kann ($t = last(p)$), hat die Eigenschaftsposition keinen Nachfolger. Die Nachfolgerabbildung ist demnach wie folgt definiert

$$succ(p^t) := \begin{cases} \{\emptyset\} & : \quad t = last(p) \\ \{\{p^{t+1}, q^{left(q)}\} \mid (p, q) \in E_{\mathcal{P}}\} & : \quad t + 1 = right(p) \\ \{\{p^{t+1}\}\} & : \quad \text{sonst.} \end{cases}$$

Das Ergebnis dieser Operation ist eine Menge von Mengen. Dieses verschachtelte Konstrukt ist notwendig, weil im Falle einer Verzweigung (mittlerer Zweig in der Formel) der um eins erhöhte Nachfolger (p^{t+1}) in jeder Verzweigung auftauchen muss. Die Eigenschaftsposition der Nachfolgereigenschaft trägt somit immer ihre Historie mit sich. Dies sind Positionen früherer Eigenschaften, die bereits ihre rechte Grenze $right(p)$ erreicht haben, aber noch nicht ihr Ende $last(p)$.

Auf Basis dieser Definition können die Nachfolger eines Zustands im nichtdeterministischen Kontrollautomaten hergeleitet werden. Dies geschieht mit Hilfe eines abgewandelten Produkts zweier Mengen von Mengen

$$M \hat{\times} N := \{m \cup n \mid m \in M, n \in N\},$$

wobei aufgrund der Kommutativität des Operators mit

$$\hat{\prod_{1 \leq i \leq n}} x_i := x_1 \hat{\times} x_2 \hat{\times} \ldots \hat{\times} x_n$$

auch ein entsprechender mehrstelliger Operator verwendet werden kann. Die Nachfolgermenge der einzelnen Eigenschaftspositionen wird nun dazu benutzt, die Nachfolgermenge eines Zustands $s \in S$ zu definieren:

$$succ(s) := \hat{\prod_{p^t \in s}} succ(p^t).$$

Dabei besteht höchstens eine der Mengen $succ(p^t)$ aus mehr als einem Element, da niemals gleichzeitig mehrere Eigenschaften ihre rechte Grenze erreichen und sich verzweigen können.

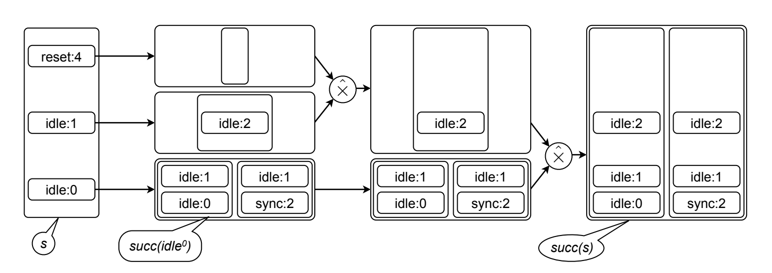


Abbildung 4.10: Entwicklung der Menge der Nachfolger eines Zustands $s \in S$.

In Abb. 4.10 ist die Berechnung der Nachfolgermenge an einem Zustand aus dem Framerbeispiel illustriert. Ausgehend von einem Zustand s, der aus drei Eigenschaftspositionen besteht, wird zuerst für jede Eigenschaftsposition die Menge der Nachfolger bestimmt. Da $reset^4$ am Ende der Eigenschaft angelangt ist, entsteht die leere Menge. Die Position $idle^1$ wird lediglich um eins erhöht, während sich $idle^0$ einen Zeitschritt vor der rechten Grenze von $idle$ befindet $(0 + 1 = right(idle))$ und deshalb nach $idle^0$ und $sync^2$ verzweigen muss. Die ursprüngliche Eigenschaftsposition wird um eins erhöht und findet sich in beiden Mengen wieder. In weiteren Schritten werden die entstehenden Mengen mit dem Operator $\hat{\times}$ verknüpft, sodass am Ende zwei Nachfolger von s entstehen.

Beispiel

Um diese recht formale Beschreibung des Konstruktionsverfahrens weiter zu verdeutlichen, soll der vollständige NEA des bereits vorgestellten Beispiels *framer* erstellt werden. Dafür muss zu den Daten in Abb. 4.7 die Menge der Annahmen nachgereicht werden. Der konkrete Ausdruck, den jede der Annahmen darstellt, ist für den Verlauf der Automatenkonstruktion nicht notwendig. Dieses Beispiel enthält lediglich vier Annahmen, die wir hier einfach durchnummerieren

$$\begin{aligned} a_0 &:= \hat{a}_{idle,0} \\ a_1 &:= \hat{a}_{sync,2} \\ a_2 &:= \hat{a}_{nosync,2} \\ a_3 &:= \hat{a}_{inframe,2}. \end{aligned}$$

Das Eingabealphabet des Automaten wird damit zu

$$\begin{aligned} \Sigma &:= 2^{\{a_0,a_1,a_2,a_3\}} \\ &:= \{\emptyset, \{a_0\}, \{a_1\}, \{a_2\}, \{a_3\}, \\ &\quad \{a_0,a_1\}, \{a_0,a_2\}, \{a_0,a_3\}, \{a_1,a_2\}, \{a_1,a_3\}, \{a_2,a_3\}, \\ &\quad \{a_0,a_1,a_2\}, \{a_0,a_1,a_3\}, \{a_0,a_2,a_3\}, \{a_1,a_2,a_3\}, \{a_0,a_1,a_2,a_3\}\}. \end{aligned}$$

Mit diesen Angaben und dem vorgestellten Konstruktionsverfahren entsteht der in Abb. 4.11 dargestellte Automat. Der Startzustand $S_0 = reset^0$ ist dabei gesondert gekennzeichnet. Die Notation in der Abbildung nutzt den Doppelpunkt zur Repräsentation eines Eigenschaftszeitpunktes ($\left[\begin{smallmatrix} reset:4 \\ sync:3 \end{smallmatrix}\right] = \{reset^4, sync^3\}$). Die für einen NEA typischen nichtdeterministischen Zustandsübergänge können bspw. im Zustand $\{sync^7\}$ betrachtet werden, der zwei ausgehende Kanten ohne zusätzliche Bedingung hat. Die Annahmen a_0 bis a_3 sind in der Abbildung durch Kreise mit dem Annahmenindex als Beschriftung dargestellt. In dieser Abbildung sind an den Übergängen nur die Minimalanforderungen σ_s angetragen, d.h. alle Kombinationen von Annahmen, in denen mindestens die Annahmen in σ_s erfüllt sind, lösen auch den Übergang aus.

Ausgaberelation

Das Ausgabealphabet Ω ist die Potenzmenge der Zusicherungen

$$\Omega := 2^Z.$$

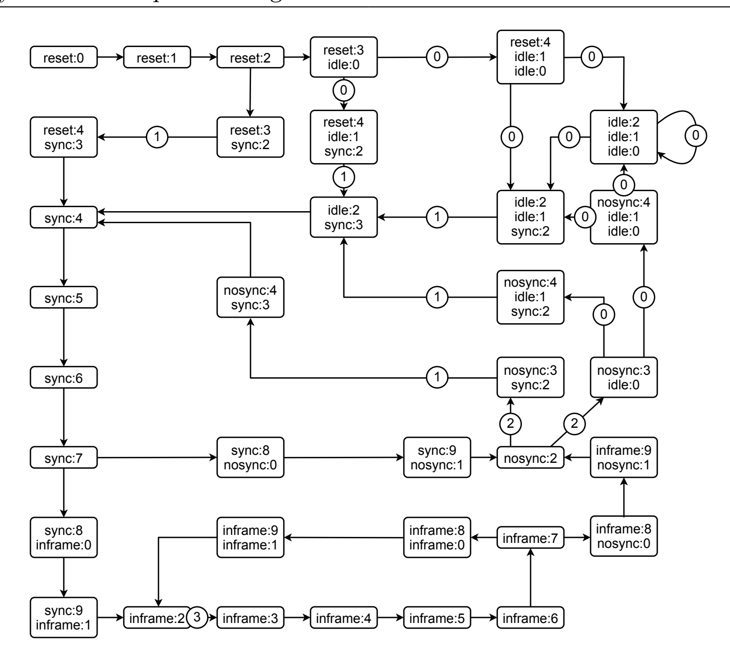


Abbildung 4.11: Nichtdeterministischer endlicher Kontrollautomat der Komponente *Framer*.

Mit Hilfe dieser Definition kann nun die Ausgaberelation $\lambda \subseteq S \times \Omega$ definiert werden. Für einen gegebenen Zustand $s \in S$ ist das gewählte Ausgabesymbol $\omega_s \in \Omega$ die Menge aller Zusicherungen $z_{p,n,t}$ (vgl. Abschnitt 4.5.6), die in einer der Eigenschaftspositionen des Zustandes gelten

$$\omega_s := \{z_{p,n,t} \mid z_{p,n,t} \in Z, p^t \in s\}.$$

Damit wird die Ausgaberelation zu

$$\lambda := \{(s, \omega_s) \mid s \in S\}.$$

4.6.4 Berücksichtigung der Erreichbarkeit und Algorithmus

Ein weiterer wichtiger Punkt ist die Erreichbarkeit der Zustände. Die formale Konstruktion des NEA definiert die Menge der Zustände des Systems mit $S = 2^Q$. Nach Beendigung des Konstruktionsverfahrens sind jedoch eine

erhebliche Anzahl dieser Zustände nicht vom Startzustand aus erreichbar. In Abb. 4.11 sind diese Zustände nicht dargestellt und es ist wünschenswert, dass sie während der Konstruktion nicht betrachtet werden, damit die Komplexität des Verfahrens möglichst gering bleibt.

Algorithmus 4.1 Funktion zum Erstellen des nichtdeterministischen Kontrollautomaten unter Berücksichtigung der erreichbaren Zustände.

1: $S \leftarrow \{S_0\}$
2: $\mathcal{Q} \leftarrow \{S_0\}$
3: **while** $\mathcal{Q} \neq \emptyset$ **do**
4:     $s \leftarrow$ wähle aus $\mathcal{Q}$
5:     $\mathcal{Q} \leftarrow \mathcal{Q} \setminus \{s\}$
6:     **for all** $s' \in succ(s)$ **do**
7:         **if** $s' \notin S$ **then**
8:             $\mathcal{Q} \leftarrow \mathcal{Q} \cup s'$
9:             $S \leftarrow S \cup s'$
10:         **end if**
11:         **for all** $\sigma \in \Sigma \mid \sigma_s \subseteq \sigma$ **do**
12:             $\delta \leftarrow \delta \cup \{(s, \sigma_s, s')\}$
13:         **end for**
14:     **end for**
15: **end while**

Aus diesem Grund wurde zu der bestehenden Konstruktionsvorschrift ein Algorithmus entwickelt, der unter Berücksichtigung der Erreichbarkeit der einzelnen Zustände den Automaten erstellt. Der passende Pseudokode ist in Alg. 4.1 dargestellt und basiert auf einer Tiefen- bzw. Breitensuche[2] des Automaten, beginnend mit dem Startzustand. Die Menge $\mathcal{Q}$ kennzeichnet dabei alle erreichbaren Zustände, die bereits gefunden, aber noch nicht bearbeitet wurden. Die Schleife in den Zeilen 11 bis 13 kann optimiert werden, da bei einer Hardwareabbildung die Übergangsbedingungen nicht auskodiert werden müssen, sondern eine Konjunktion der Annahmen in σ_s ausreicht.

In Alg. 4.2 ist der Pseudokode für die Berechnung der Funktion $succ(s)$ abgebildet. Die Zeilen 3 bis 12 berechnen die Funktion $succ(p^t)$ für jede Eigenschaftsposition p^t in s. Das Resultat wird in der Variable m gespeichert und in den Zeilen 14 bis 20 mit dem Zwischenergebnis s' multipliziert (entspre-

[2] Je nach Implementierung kann in Zeile 4 entweder das zuletzt oder das zuerst zu $\mathcal{Q}$ hinzugefügte Element gewählt werden. Ersteres führt zu einer Tiefensuche, während letzteres zu einer Breitensuche führt.

Algorithmus 4.2 Eine mögliche Implementierung der Nachfolgerfunktion $succ(s)$.

```
1: s' ← {∅}
2: for all p^t ∈ s do
3:    if t + 1 = right(p) then
4:       m ← ∅
5:       for all q ∈ P | {p, q} ∈ E_𝒫 do
6:          m ← m ∪ {{p^{t+1}, q^{left(q)}}}
7:       end for
8:    else if t + 1 ≤ last(p) then
9:       m ← {{p^{t+1}}}
10:   else
11:      m ← {∅}
12:   end if
13:
14:   s'' ← {∅}
15:   for all s'_i ∈ s' do
16:      for all m_i ∈ m do
17:         s'' ← s'' ∪ {s'_i ∪ m_i}
18:      end for
19:   end for
20:   s' ← s''
21: end for
22: return s'
```

chend der Operation $s' \hat{\times} m$). Nach Abarbeitung aller Eigenschaftspositionen wird s' als Endergebnis zurückgegeben.

Der Algorithmus von $succ(s)$ kann vereinfacht werden, indem diejenige Eigenschaftsposition, an der möglicherweise eine Verzweigung eintritt (Zeile 4 bis 6) in der letzten Iteration behandelt wird. In diesem Fall hat m für alle anderen Eigenschaftspositionen jeweils nur ein Element und die beiden Schleifen (Zeilen 15 bis 19) werden je nur einmal durchlaufen. Lediglich in der letzten Iteration der Hauptschleife kann m dann mehrere Elemente besitzen und Zeile 17 wird exakt $|m|$ mal ausgeführt.

4.6.5 Komplexitätsbetrachtung

Die Vorstellung eines Algorithmus ist nur dann vollständig, wenn auch eine Abschätzung seiner Komplexität unternommen wird. So muss zum einen der Speicherbedarf als auch die zu erwartende Laufzeit untersucht werden. Der

Speicherbedarf richtet sich hauptsächlich nach der Anzahl der zu untersuchenden Zustände, weshalb sich im folgenden Unterabschnitt darauf konzentriert wird.

Speicher

Die Frage nach der Komplexität, bzw. der Anzahl erreichbarer Zustände, des NEA lässt sich nicht exakt beantworten, da sie in hohem Maße von der Form der Eigenschaften abhängt. Eine triviale obere Schranke kann angegeben werden, indem die Kardinalität der Potenzmenge bestimmt wird

$$|S| := 2^{|Q|}.$$

Diese Anzahl Zustände wäre jedoch in keiner Weise beherrschbar, sodass eine realistischere Abschätzung benötigt wird.

Die praktische Erfahrung hat gezeigt, dass die Anzahl der Zustände des NEA linear mit der Anzahl der Zustände in Q, d.h. mit der Anzahl der Eigenschaftspositionen, wächst. Das heisst im Umkehrschluss auch, dass die Zustände sowohl linear mit der Anzahl der Eigenschaften, als auch deren Länge wachsen.

Die Ausnahme von dieser Regel bilden Eigenschaften, die eine starke Überlappung aufweisen. Dies tritt insbesondere bei kurzen Eigenschaften mit hoher Pipelinelänge auf. Formal ausgedrückt kann ein Überlappungsfaktor

$$u(p) := \left\lceil \frac{last(p) - left(p) + 1}{length(p)} \right\rceil$$

errechnet werden, der angibt wie umfangreich die Historie einer Eigenschaft ist, d.h. wieviele ältere Eigenschaften während der Abarbeitung einer aktiven Eigenschaft noch nicht ihren letzten Zeitpunkt erreicht haben.

Wenn bspw. eine Eigenschaft für die Attribute $left(p)$, $right(p)$ und $last(p)$ die Werte 0, 4 und 19 besitzt, ist ihr Überlappungsfaktor $u = 5$, d.h. wenn diese Eigenschaft auf sich selbst folgen kann ($(p, p) \in E_{\mathcal{P}}$), besitzt ein Zustand in S maximal fünf Eigenschaftspositionen, d.h. es kann eine Überlappung von maximal fünf Instanzen der Eigenschaft geben.

Im Falle von drei Eigenschaften gleicher Länge, die jede auf jede andere folgen können, kann diese Historie der Länge fünf an jeder Position eine der drei Eigenschaften aufweisen. Weiterhin tritt eine Erhöhung der Eigenschaftspositionen über vier Takte ein ($length(p) = 4$), wobei danach die nächste Verzweigung stattfindet und damit wieder die Ausgangssituation erreicht ist. Die Anzahl der Zustände kann mit $4 \cdot 3^5 = 972$ abgeschätzt werden. Die tatsächliche Anzahl der Zustände, die *vhisyn* für dieses Beispiel erzeugt, beträgt 1132, was in der gleichen Größenordnung liegt.

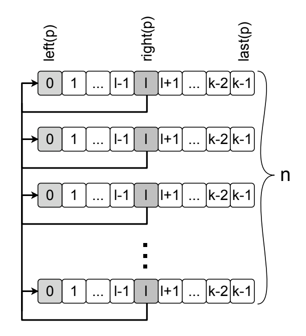


Abbildung 4.12: Abgerollter Eigenschaftsgraph von n gleichartigen Eigenschaften mit vollständiger Verknüpfung.

Der generelle Fall, der von diesem Beispiel abgeleitet werden kann, ist eine Anzahl von n gleichartigen Eigenschaften, die mit $left(p) = 0$, $right(p) = l$ und $last(p) = k - 1$ ihre Grenzen definieren. In dem vollständigen Eigenschaftsgraph kann jede Eigenschaft kann auf jede andere folgen. In Abb. 4.12 ist ein solches Szenario in Form des entsprechenden abgerollten Eigenschaftsgraphen dargestellt. Die Zeitkanten sind nicht dargestellt.

Die exakte Anzahl der Zustände für den abgeleiteten NEA kann in diesem Fall durch die Formel

$$|S| = (n^u - n^{u-1} + n^{u-2})(\frac{l \cdot n}{n-1} + k - l \cdot u) - \frac{l}{n-1}$$

angegeben werden. Auf das vorgenannte Beispiel angewandt, berechnet diese Formel den Wert 1132. Der massgebliche Anteil der Formel ist $l \cdot n^u$ (für große n und $k \bmod l = 0$), womit er in die Komplexitätsklasse

$$|S| \in \mathcal{O}(l \cdot n^u)$$

fällt.

Davon lässt sich ein allgemeinerer Fall ableiten. Der Eigenschaftsgraph ist nicht vollständig, besteht jedoch weiterhin aus gleichartigen Eigenschaften. Dann ist nicht mehr die Anzahl der Eigenschaften für die Komplexitätsbetrachtung interessant, sondern die Anzahl der Verzweigungen, d.h. die ausgehenden Kanten $deg^+(\mathcal{P})$ im Eigenschaftgraph. Die Komplexitätsklasse definiert sich damit zu

$$|S| \in \mathcal{O}\left(l \cdot n \cdot \left(deg^+(\mathcal{P})\right)^u\right),$$

was bei einer großen Anzahl gering verknüpfter Eigenschaften eine deutlich verbesserte Abschätzung der Anzahl Zustände bietet. Diese Formel gilt nur bei gleichartigen Eigenschaften in einem symmetrisch verknüpften Eigenschaftsgraph. Für unreguläre Strukturen sind diese Abschätzungen mathematisch nicht mehr gültig, jedoch hat die Praxis gezeigt, dass die Anzahl der Zustände im Wesentlichen dieser Formel folgt.

Zusammenfassend können folgende Behauptungen über die Anzahl der Zustände des NEA aufgestellt werden:

- Die *Anzahl der Nachfolger* einer Eigenschaft hat *polynomiellen* Einfluss auf die Anzahl der Zustände.
- Der *Überlappungsfaktor* hat *exponentiellen Einfluss*,
- während sich die *Länge* und *Anzahl* der Eigenschaften *linear* auf die Anzahl Zustände auswirkt.

Laufzeit

Die Laufzeit des Verfahrens orientiert sich im Wesentlichen an der Anzahl der Zustände. Die Laufzeit der Nachfolgerberechnung $succ(s)$ wächst linear mit dem Überlappungsfaktor u (entspricht $|s|$) und der Anzahl der Verzweigungen $deg^+(\mathcal{P})$, sodass dieser Faktor gegenüber der Anzahl Zustände nicht ins Gewicht fällt. Insgesamt lässt sich näherungsweise sagen, dass die Komplexität der Laufzeit sich in einem ähnlichen Rahmen wie die Komplexität der Anzahl Zustände bewegt.

4.7 Konstruktion eines deterministischen Kontrollautomaten

Die in diesem Abschnitt beschriebenen Verfahren wurden ursprünglich im Rahmen der Arbeit von Pepelyashev [Pep09] entwickelt und wurden erstmals in [LPH10] veröffentlicht. Obwohl die Funktionsweise des Algorithmus weitestgehend erhalten geblieben ist, unterscheidet sich die Form der Darstellung von den angegebenen Veröffentlichungen.

Bei der Konstruktion des NEA wurden alle Annahmen lediglich "durchnummeriert" und als Übergangsbedingungen an den Automaten annotiert. Dies ist im nichtdeterministischen Fall völlig ausreichend, da vor der Synthese der Vollständigkeitsbeweis des Eigenschaftssatzes stattgefunden hat. Unter Annahme der in Abschnitt 3.6.2 eingeführten Abwandlung des Fallunterscheidungstests ist damit sichergestellt, dass Eigenschaften zwar eine Zeit

lang konkurrieren dürfen, aber letztendlich immer eine eindeutige Entscheidung zugunsten einer einzigen Eigenschaft fallen muss und dass das Verhalten vor dem Eintritt der eindeutigen Entscheidung in allen konkurrierenden Eigenschaften gleich sein soll.

Durch diese Einschränkungen ist sichergestellt, dass der NEA eine nichtdeterministische Entscheidung (mehrere parallele Eigenschaften sind gleichzeitig aktiv) nach einer gewissen Zeit wieder auflöst. Spätestens am Ende einer Eigenschaft muss jedoch feststehen, ob sie aktiviert wurde oder nicht. Falls ja, darf gleichzeitig keine weitere parallele Eigenschaft aktiviert worden sein. Diese Auflösung des Nichtdeterminismus ist durch die Eigenschaften nur implizit gegeben und durch die Vollständigkeitstests geprüft.

Bei der nichtdeterministischen Konstruktion werden die "durchnummerierten" Annahmen als völlig unabhängig voneinander betrachtet. Dadurch hat man den Vorteil eines schnellen und effektiven Konstruktionsalgorithmus, den man sich durch den Nichtdeterminismus des entstehenden Automaten erkauft. Wie in Abschnitt 4.9.2 beschrieben wird, existiert ein Verfahren diesen NEA in eine effektive Hardwarebeschreibung zu überführen [SP01; Sie05]. Dabei wird im Wesentlichen das Verhalten des Automaten "emuliert" und die Aktivierung von jedem Zustand in einem eigenen Register gespeichert. Der Nachteil des Verfahrens ist der hohe Bedarf an Registern, der der Anzahl an Zuständen des Automaten entspricht.

Der Aufbau dieses Abschnitts folgt nicht der Reihenfolge des Synthesevorgangs. Es wird zuerst die Umwandlung des NEA in einen DEA mit Hilfe des Potenzmengenverfahrens vorgestellt, um davon ausgehend ein verbessertes Automatenkonstruktionsverfahren zu entwickeln. Das Werkzeug *vhisyn* nutzt diesen Umweg nicht. Es erzeugt direkt einen DEA, ohne vorher den NEA oder gar das Potenzmengenverfahren anzuwenden. Weiterhin bietet *vhisyn* die Möglichkeit je nach Anwendungsfall zwischen der nichtdeterministischen und der deterministischen Konstruktion zu wählen.

4.7.1 Konstruktion des Potenzmengenautomaten

Es ist vorteilhaft, ein alternatives Konstruktionsverfahren für den Kontrollautomaten zu entwickeln, welches einen deterministischen endlichen Automaten (DEA) erzeugt, da dessen Hardwarerepräsentation mit Hilfe einer binären Automatenkodierung wesentlich weniger Register benötigt als der entsprechende NEA. Das Standardverfahren der Literatur für diesen Fall heißt Potenzmengenverfahren [HMU06] und ist an einem Beispiel in Abb. 4.13 illustriert. Da das Beispiel aus Abb. 4.9 durch die Potenzmengenkonstruktion zu groß für eine übersichtliche Darstellung in dieser Arbeit werden würde, ist ein noch weiter vereinfachter Automat dargestellt. Der Automat erkennt

jeweils in einer fortlaufenden Zeichenkette die Sequenz "cat". Die unbedingte Transition von Zustand 1 zu sich selbst sorgt dafür, dass der Suchvorgang mit jedem Eingangssymbol neu gestartet wird und der Automat nichtdeterministisch ist.

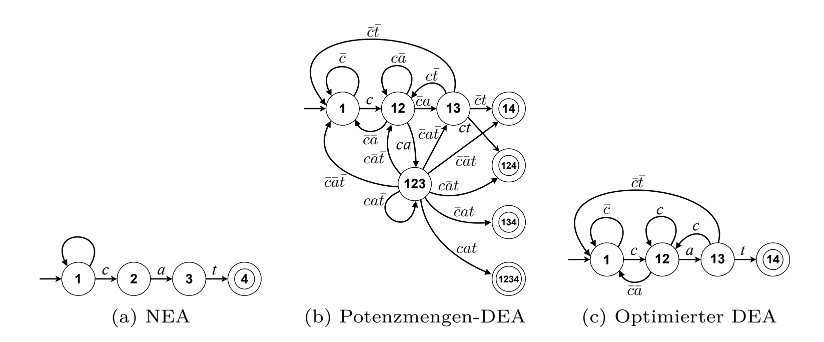


Abbildung 4.13: Beispiel eines endlichen Automaten zur Erkennung der Zeichenkette "cat" in einem Text.

Die Knoten des Potenzmengenautomat (PMA) repräsentieren jeweils eine Menge von Zuständen des NEA. Wenn sich der PMA im Zustand 12 befindet (und da es ein DEA ist, kann kein anderer Zustand gleichzeitig aktiviert sein), bedeutet dies, dass sich der entsprechende DEA sowohl in Zustand 1 als auch in Zustand 2 befindet. Im schlimmsten Fall (alle Zustände sind erreichbar) hat der PMA demnach $2^{|NEA|}$ Zustände.

Die Übergangsbedingungen müssen bei der Auflösung des Determinismus komplett auskodiert werden, d.h. für jede Kombination von Übergangsbedingungen, die auf ausgehenden Kanten eines der entsprechenden Zustände des NEA auftreten, muss eine neue Kante im PMA hinzugefügt werden. So muss im Beispiel der Zustand 12 alle ausgehenden Kanten von 1 und von 2 berücksichtigen. Diese sind

$$\{(1, 1, 1), (1, c, 2), (2, a, 3)\} \in \delta_{NEA},$$

wobei die Tupel wie gewohnt (Startzustand, Übergangsbedingung, Endzustand) repräsentieren. Der bedingungslose Übergang ist durch eine 1 (logisch wahr) an zweiter Stelle des Tupels gekennzeichnet. Die beiden Übergangsbedingungen c und a erlauben nun wieder 4 ausgehende Kanten im DEA

$$\{(12, \bar{c}\bar{a}, 1), (12, c\bar{a}, 12), (12, \bar{c}a, 13), (12, ca, 123)\} \in \delta_{PMA}.$$

4.7.2 Erreichbarkeit im Potenzmengenautomaten

Die Konstruktion des vorgestellten Beispiel-PMA illustriert einen wichtigen Punkt. Viele der Kanten im PMA sind nicht erreichbar, denn bei exakter Betrachtung fällt auf, dass diese Kanten mehrere gleichzeitige Aktivierungen der originalen Übergangsbedingungen erfordern, d.h. die Kante $(12, ca, 123)$ benötigt in der Eingangszeichenkette sowohl das Zeichen c als auch das Zeichen a. Da aber immer nur genau ein Zeichen, entweder c oder a, auftritt, wird dieser Übergang niemals aktiviert. Beide Teilbedingungen sind nicht unabhängig voneinander. Wenn dieser Fakt bei der Konstruktion in Betracht gezogen wird, können alle Übergänge, die das Auftreten mehrerer Zeichen erfordern, entfallen und der Automat vereinfacht sich stark. In der Abb. 4.13c ist das Resultat als optimierter DEA dargestellt.

Auf die Konstruktion eines DEA für den Kontrollautomaten der Eigenschaftsbeschreibung bezogen, bedeutet dies, dass bei der Konstruktion des DEA beachtet werden muss, welche Annahmen sich gegenseitig ausschliessen. Das ist prinzipiell durch die Formulierung eines Erfüllbarkeitsproblems (SAT) und die anschließende Prüfung mittels eines SAT-Solvers möglich [DP60; DLL62]. Die Prüfung kann jedoch unter Umständen sehr aufwendig werden, da die Lösung des aussagenlogischen Erfüllbarkeitsproblems im Allgemeinen NP-vollständig ist [Kar72]. Eines der hauptsächlichen Ziele dieser Arbeit ist jedoch die Vermeidung von Erfüllbarkeitsproblemen, da die Laufzeit des Synthesewerkzeugs dadurch weniger vorhersehbar wird und insbesondere komplexe Zusammenhänge zwischen Annahmen zu hohem Prüfaufwand führen.

4.7.3 Klassifikation der Annahmen

Es können auch ohne die mathematische Prüfung der Unabhängigkeit mehrerer Annahmen einige wichtige Aussagen getroffen werden, die sich direkt aus der Struktur der Eigenschaften und des Eigenschaftsgraphen herleiten. Dazu müssen die Gesamtannahmen in zwei Gruppen unterteilt werden. Im weiteren Verlauf wird vereinfachend von Annahmen anstelle von Gesamtannahmen gesprochen. Zum einen wird die letzte Annahme einer jeden Eigenschaft als *abschliessende Annahme* bezeichnet und in die Menge

$$A_F := \left\{ \hat{a}_{p,i} \in \hat{A} \mid \left(\nexists\, \hat{a}_{p,j} \in \hat{A} : j > i \right) \right\}$$

eingetragen. Zum anderen werden alle nicht zu A_F zählenden Annahmen als Zwischenannahmen

$$A_I := \hat{A} \setminus A_F$$

bezeichnet.

4.7.4 Kriterien zur Auflösung des Nichtdeterminismus

Diese Unterscheidung hat den Hintergrund, dass bei der Auswertung von beiden Annahmetypen unterschiedlich vorgegangen werden muss. Während die Ablehnung jeder Annahme auch die Nichtaktivierung der gesamten Eigenschaft nach sich zieht, reicht eine positive Zwischenannahme nicht für eine endgültige Aktivierung der Eigenschaft. Eine positive abschliessende Annahme ist jedoch eine hinreichende Bedingung für die Aktivierung.

Weiterhin zieht die endgültige Aktivierung einer Eigenschaft die automatische Deaktivierung aller parallel verlaufenden Eigenschaften und deren Nachfolger nach sich. Die gewonnenen Informationen können in fünf Kriterien zusammengefasst werden:

1. *Nicht erfüllte Zwischenannahmen* führen zur Beendigung der Eigenschaft und ihrer Nachfolger.

2. *Nicht erfüllte abschliessende Annahmen* führen ebenfalls zur Beendigung der Eigenschaft und ihrer Nachfolger.

3. *Erfüllte Zwischenannahmen* haben keine Auswirkung.

4. *Erfüllte abschliessende Annahmen* führen zur Beendigung aller parallelen Eigenschaften und ihrer Nachfolger.

5. Es muss immer *mindestens eine Eigenschaft* aktiviert sein.

Während die Kriterien 1 bis 3 selbsterklärend sein sollten, ist eine genauere Definition der parallelen Eigenschaften in Kriterium 4 schwieriger. Eine Erklärung gelingt am besten mit Hilfe eines Zustandes aus dem *Framer* Beispiel (siehe Abb. 4.11). Die Annahmen a_0 und a_1 sind ebenfalls dem Beispiel entnommen. Der Zustand $\{nosync^3, idle^0\}$ hat einen nichtdeterministischen Übergang in die Zustände $\{nosync^4, idle^1, sync^2\}$ und $\{nosync^4, idle^1, idle^0\}$. Bei diesem Übergang entspricht die Eigenschaftsposition $nosync^4$ dem Nachfolger von $nosync^3$ (Zeitkante entsprechend Abschnitt 4.6.1). In Abb. 4.14 kann ausgehend von $\{nosync^3, idle^0\}$ der Ablauf der Eigenschaften für drei Zeitschritte betrachtet werden. Zur selben Instanz einer Eigenschaft gehörende Eigenschaftspositionen sind entsprechend markiert und verbunden.

Die Zustände $\{nosync^4, idle^1, sync^2\}$ und $\{nosync^4, idle^1, idle^0\}$ werden als parallel bezüglich $sync^2$ oder $idle^0$ bezeichnet. Bezüglich $idle^1$ und $nosync^4$ sind die beiden Zustände nicht parallel, da die entsprechende Verzweigung zu einem früheren Zeitpunkt eingetreten war. Laut Kriterium 4 bedeutet die Erfüllung der abschliessenden Annahme a_0 in allen ausgehenden

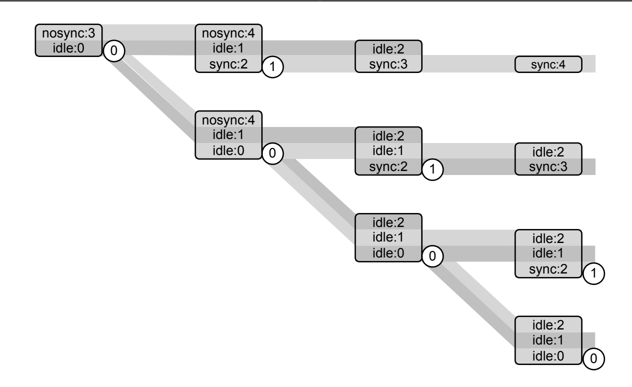


Abbildung 4.14: Verlauf von Eigenschaften über mehrere Zustände des NEA am Beispiel der Komponente *Framer*.

Kanten von $\{nosync^4, idle^1, idle^0\}$ die automatische Beendigung der parallelen Zustände bezüglich a_0, d.h. der Zustand $\{nosync^4, idle^1, sync^2\}$ kann in diesem Fall nicht weiter beachtet werden. Die Annahme a_1 muss demnach negativ werden. Der beschriebene Sachverhalt lässt sich allerdings auch umkehren. Wenn die abschliessende Annahme a_1 erfüllt ist, wird der Zustand $\{nosync^4, idle^1, idle^0\}$ beendet, bzw. alle seine ausgehenden Kanten sind ungültig.

Es bleibt nun die Frage, wie die Annahmen an die entstehenden Kanten des DEA annotiert werden. So könnte man die Annahme a_1 durch $\bar{a_0}$ ersetzen, oder durch die auskodierte Form $\bar{a_0}a_1$. Unter Ausnutzung der oben aufgeführten Kriterien ist idealerweise diejenige Möglichkeit zu wählen, die mit der geringsten, d.h. ressourcenschonendsten und schnellsten, Hardwareabbildung einhergeht.

Es erscheint nun ausreichend alle Übergänge zu verbieten, in denen mehrere abschliessende Annahmen gleichzeitig auftreten. Das ist jedoch nicht der Fall. Wenn bspw. Annahme a_1 nicht der Eigenschaftsposition $sync^2$ sondern $sync^3$ zugeordnet ist, wird Zustand $\{idle^2, sync^3\}$ fälschlicherweise aktiviert, obwohl die eindeutige Entscheidung bereits zu Gunsten des anderen Pfades gefallen ist. Die um einen Takt später angesetzte Annahme muss bei erfolgreich durchgeführtem Vollständigkeitstest, auf jeden Fall negativ ausgewertet werden. Jedoch können unterschiedliche Zusicherungen in $\{idle^2, sync^3\}$ gegenüber $\{idle^2, idle^1, sync^2\}$ und $\{idle^2, idle^1, idle^0\}$ bereits unerwünschtes Systemverhalten hervorrufen.

An dieser Stelle muss darauf hingewiesen werden, dass die Parallelität von $\{idle^2, idle^1, sync^2\}$ und $\{idle^2, idle^1, idle^0\}$ nicht aufgelöst werden kann, da die entsprechende Annahme erst in diesem Schritt ausgewertet wird. Es ist also notwendig, dass $sync^2$ und $idle^0$ keine sich widersprechenden Zusicherungen enthalten.

4.7.5 Formales Konstruktionsverfahren

Partielle Ordnung der Eigenschaftpositionen

Ausgehend von dieser Definition der Kriterien kann nun der eigentliche Konstruktionsalgorithmus vorgestellt werden. Dazu ist es nötig eine partielle Ordnung in der Menge der Eigenschaftspositionen zu definieren. Die geordnete Menge $(Q, \prec)$ mit der Ordnungsrelation

$$p^t \prec q^u := (t - right(p)) < (u - right(q))$$

sorgt dafür, dass die "jüngeren" Eigenschaftspositionen weiter vorn angeordnet werden, während die älteren Eigenschaften weiter hinten vorkommen. Es ist dabei zu beachten, dass es genau eine Eigenschaftsposition p^t in jedem Zustand $s \in S$ gibt, für die $(t - right(p)) < 0$ ist, d.h. sie ist die gegenwärtig aktive Eigenschaft.

DEA als 6-Tupel

Der zu konstruierende deterministische endliche Automat wird durch ein 6-Tupel

$$\mathcal{A}_{DEA} = (V, V_0, \Sigma, \Omega, \delta', \lambda')$$

charakterisiert, bei dem

- V die Menge der Zustände,
- V_0 der Startzustand $V_0 \in V$,
- Σ das Eingabealphabet,
- Ω das Ausgabealphabet,
- δ' die Zustandsübergangsrelation $\delta' \subseteq V \times \Sigma \times V$,
- sowie λ' die Ausgaberelation $\lambda' \subseteq V \times \Omega$

sind. Damit ist der Automat ebenso wie der NEA ein Moore-Automat, bei dem die Ausgaberelation nicht vom Eingabealphabet abhängt.

Zustände

Die Zustände $v \in V$ des neuen deterministischen Automaten entstammen der Potenzmenge von S, mit

$$V := 2^S.$$

Die Elemente $s \in S$ eines Zustands $v \subseteq S$, die gleichzeitig den Zuständen des NEA entsprechen, werden im folgenden auch als Pfade bezeichnet. Der Startzustand ist ähnlich dem Startzustand des NEA als Menge

$$V_0 := \{S_0\}$$

definiert, dessen einziges Element S_0 ist, was wiederum dem Anfang der Reseteigenschaft entspricht.

Eingabealphabet

Das Eingabealphabet Σ ist das selbe wie das Eingabealphabet des NEA. Die Elemente von Σ sind dabei beliebige Teilmengen aller Gesamtannahmen

$$\Sigma := 2^{\hat{A}}.$$

Für ein gegebenes Eingabesymbol $\sigma \in \Sigma$, das exakt alle erfüllten Annahmen enthält, kann die Menge der nicht erfüllten Annahmen durch

$$\bar{\sigma} := \hat{A} \setminus \sigma$$

ausgedrückt werden.

Parallele Zustände

Für die Definition der Zustandsübergangsrelation ist zuerst eine formale Bestimmung des parallelen Zustandsbegriffs aus Kriterium 4 notwendig. Gegeben sind zwei Pfade (Zustände des NEA) $s_1, s_2 \in S$. Nun gilt s_2 parallel zu s_1 bezüglich einer Eigenschaftsposition $p^t \in s_1$, wenn alle älteren Eigenschaftspositionen als p^t in beiden Zuständen gleich sind und aber die Eigenschaftsposition p^t nicht in s_2 vorkommt

$$parallel(s_1, p^t, s_2) := \big(prefix(s_1, p^t) = prefix(s_2, p^t)\big) \wedge p^t \notin s_2,$$

wobei die Hilfsfunktion $prefix$ mit

$$prefix(s, p^t) := \{q^u \in s \mid p^t \prec q^u\}$$

definiert wird und alle "älteren" Eigenschaftspositionen sammelt. Die Bedingung $p^t \notin s_2$ verhindert, dass ältere Verzweigungen nicht verworfen werden, sondern nur Verzweigungen bezüglich p^t.

Zustandsübergangsrelation

Die Zustandsübergangsrelation ergibt sich wie folgt

$$delta' := \{(v, \sigma, succ(v, \sigma)) \mid v \in V, \sigma \in \Sigma, succ(v, \sigma) \neq \emptyset\}.$$

Dabei wird wieder eine Nachfolgerfunktion $succ(v, \sigma)$ benötigt, die in diesem Fall aber ein zweites Argument in Form der Übergangsbedingung besitzt. Die Bedingung $succ(v, \sigma) \neq \emptyset$ stellt sicher, dass der Folgezustand nicht die leere Menge darstellt. Dadurch wird Kriterium 5 garantiert und die leere Menge ist kein erreichbarer Zustand von V.

Die Bestimmung der Nachfolger geschieht in zwei Schritten. Im ersten Schritt werden für einen Zustand $v \in V$ und eine gegebene Übergangsbedingung $\sigma \in \Sigma$ die für den Übergang relevanten Pfade $v_\sigma \subseteq v$ bestimmt. Diese werden im zweiten Schritt benutzt, um $succ(v, \sigma)$ zu definieren.

Zuerst wird für die Kriterien 1 bis 4 eine Bedingung $K^i_{s,\sigma}$ angegeben, die für einen bestimmten Pfad $s \in v$ festlegt, ob dieser zu den relevanten Pfaden gehört. Die Funktion $K^1_{s,\sigma}$ gibt wahr zurück, wenn für alle nicht erfüllten Zwischenannahmen die passende Eigenschaftsposition nicht in s vorkommt

$$K^1_{s,\sigma} := \forall \hat{a}_{p,t} \in (A_I \cap \bar{\sigma}) : p^t \notin s.$$

Kriterium 2 wird für die abschließenden Annahmen analog definiert mit

$$K^2_{s,\sigma} := \forall \hat{a}_{p,t} \in (A_F \cap \bar{\sigma}) : p^t \notin s,$$

während Kriterium 3 trivialerweise immer zum Wahrheitswert 1 ausgewertet wird

$$\begin{aligned} K^3_{s,\sigma} \quad &:= \quad \forall \hat{a}_{p,t} \in (A_I \cap \sigma) : 1 \\ &:= \quad 1 \end{aligned}$$

Eine weitere Funktion $K^4_{s,\sigma}$ stellt sicher, dass es keinen anderen Pfad $s' \in v$ gibt, der eine Eigenschaftsposition mit abschliessender erfüllter Annahme besitzt, sodass s parallel zu s' ist

$$K^4_{s,\sigma} := \forall \hat{a}_{p,t} \in (A_F \cap \sigma) : \left(\nexists s' \in v : (p^t \in s') \wedge parallel(s', p^t, s)\right).$$

Mit Hilfe dieser vier Funktionen kann die Menge der für σ zu beachtenden Pfade v_σ definiert werden

$$v_\sigma := \{s \in v \mid K^1_{s,\sigma} \wedge K^2_{s,\sigma} \wedge K^3_{s,\sigma} \wedge K^4_{s,\sigma}\}.$$

Die eingangs benutzte Funktion $succ(v, \sigma)$ wird ihrerseits definiert durch die Vereinigung der Menge der Nachfolger jedes einzelnen Elements aus v_σ

$$succ(v, \sigma) := \bigcup_{s}^{v_\sigma} succ(s).$$

Beispiel

In Abb. 4.15 ist der nach diesem Verfahren gebildete deterministische Automat des *Framer* Beispiels dargestellt. Es sind ebenso wie im NEA aus Abb. 4.11 nur die positiven Annahmen dargestellt. Nicht gezeigte Annahmen sind im Ausgangszustand nicht relevant, d.h. der Zustandsübergang ist für erfüllte und nicht erfüllte Annahmen gleich.

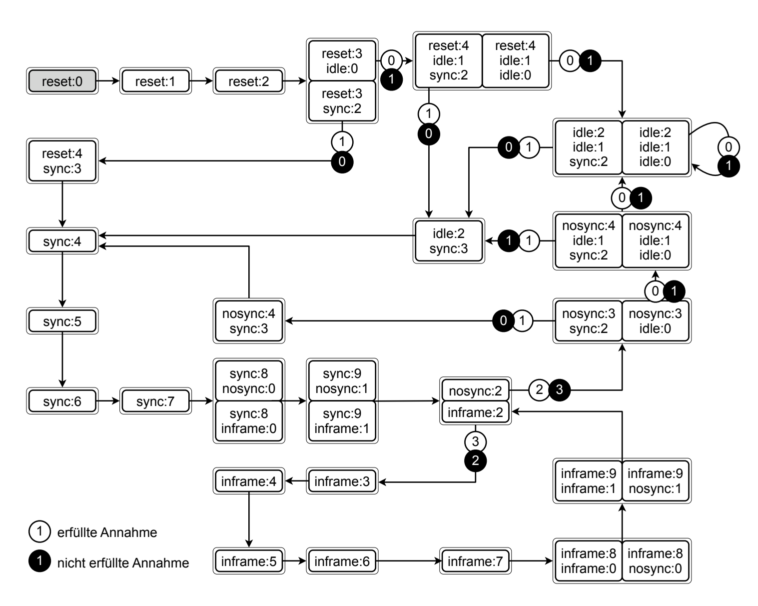


Abbildung 4.15: DEA der Komponente *Framer*.

Es kann beobachtet werden, dass der DEA in diesem Fall sogar weniger Zustände besitzt als der entsprechende NEA. Davon kann jedoch im allgemeinen Fall nicht ausgegangen werden.

Ausgaberelation

Das Ausgabealphabet ist ebenso wie für den NEA die Potenzmenge der Zusicherungen

$$\Omega := 2^Z.$$

Nun ergibt sich die Ausgaberelation $\lambda' \subseteq V \times \Omega$ zu

$$\lambda' := \{(v, \omega_v) \mid v \in V\},$$

wobei $\omega_v \in \Omega$ alle Zusicherungen enthält, die für den gegebenen Zustand v zuständig sind und vom Gesamtsystem eingehalten werden müssen

$$\omega_v := \{z_{p,n,t} \in Z \mid \exists_s^v p^t \in s\}.$$

Es dürfen mehrere Zusicherungen, die das selbe Ausgangssignal n betreffen, in ω_v vorkommen, diese müssen jedoch entweder aufgrund der erfolgreichen Vollständigkeitsprüfung den selben Wert zuweisen oder anderweitig das selbe Systemverhalten hervorrufen. Die in Abschnitt 5.1.4 zu definierenden Anforderungen an eine synthesegerechte ITL-Beschreibung sind dabei zu beachten. Welche der parallelen Zusicherungen auf das gleiche Signal von der generierten Hardware ausgeführt wird, ist implementierungspezifisch. Es ist jedoch wünschenswert den Benutzer in dieser Situation auf das gegebenenfalls redundante oder sich widersprechende Verhalten hinzuweisen.

Wenn dennoch ein Satz von Eigenschaften synthetisiert werden soll, der sich widersprechende Zusicherungen im gleichen Zustand enthält und dadurch nicht realisierbar ist, dann erzeugt die Synthese ein Design, welches sich willkürlich für eine der sich widersprechenden Zusicherungen entscheidet. Der Fehler wird entdeckt, wenn die ursprünglichen Eigenschaften auf dem erzeugten Entwurf formal geprüft werden.

4.7.6 Erreichbarkeit und Algorithmus

Die Erreichbarkeit der Zustände im DEA kann, genau wie die Erreichbarkeit des NEA, durch einen rekursiven Konstruktionsalgorithmus sichergestellt werden. Dazu wird Alg. 4.1 mit geringen Abweichungen (bedingt durch die geänderte Definition der Zustände) wiederverwendet. In Alg. 4.3 ist der entsprechende Pseudokode dargestellt.

Ein wichtiger Unterschied ist allerdings die Berechnung der Nachfolger eines Zustandes $v \in V$. Die Funktion bekommt einen zusätzlichen Parameter σ, weshalb vor dem Aufruf erst über alle Elemente in Σ iteriert werden muss. Der entsprechende Pseudokode der Nachfolgerberechnung ist in Alg. 4.4 dargestellt. Im ersten Abschnitt des Algorithmus (Zeilen 1 bis 15) werden die relevanten Pfade von v bezüglich des Übergangsbedingung σ gefiltert, wobei Zeilen 4 bis 5 die Kriterien 1 und 2 bearbeitet, während Zeilen 6 bis 12 Kriterium 4 behandeln. In Zeile 15 wird letztendlich Sorge getragen, dass Kriterium 5 eingehalten wird. Im letzten Abschnitt (Zeilen 19 bis 22) wird mit Hilfe von $succ(s)$ aus Alg. 4.2 die Nachfolgermenge zusammengefügt und anschliessend zurückgegeben.

Algorithmus 4.3 Funktion zum Erstellen des deterministischen Kontrollautomaten unter Berücksichtigung der erreichbaren Zustände.

```
V ← {V_0}
Q ← {V_0}
δ' ← ∅
while Q ≠ ∅ do
  v ← wähle aus Q
  Q ← Q \ {v}
  for all σ ∈ Σ do
    v' ← succ(v, σ)
    if v' ≠ ∅ then
      if v' ∉ V then
        Q ← Q ∪ v'
        V ← V ∪ v'
      end if
      δ' ← δ' ∪ {(v, σ, v')}
    end if
  end for
end while
```

4.7.7 Komplexitätsbetrachtung

Der zu erwartende Speicherbedarf des DEA wächst mit der Anzahl der nicht auflösbaren (bzw. spät auflösbaren) Entscheidungen. In praktischen Anwendungsbeispielen werden diese Fälle nicht zu einer übermässig erhöhten Anzahl Zustände führen, da es dann ratsam ist auf die deterministische Konstruktion zu verzichten und statt dessen den nichtdeterministischen Automaten zu erstellen.

Als bedeutsamster Faktor kann die Kardinalität der Potenzmenge der Annahmen betrachtet werden. Mit jeder zusätzlichen Annahme verdoppelt sich die Anzahl der Zustandsübergänge in $\mathcal{A}_{DEA}$. Die Mehrzahl der Übergänge kann jedoch wieder zusammengefasst werden, da Start- und Zielzustand identisch sind. Die Größe der Disjunktion der Übergangsbedingungen, von denen jede einen Minterm der Annahmen darstellt, wird dann ebenso wie zuvor die Anzahl der Übergänge drastisch anwachsen und müsste mit booleschen Optimierungsverfahren minimiert werden. Da ein modernes Synthesetool die Funktionalität für derartige Optimierungen bereits in sehr weit entwickelter Form besitzt, werden in *vhisyn* keine diesbezüglichen Anstrengungen unternommen und alle Übergänge werden entsprechend Alg. 4.3 direkt in Hardware abgebildet.

Algorithmus 4.4 Eine mögliche Implementierung der Nachfolgerfunktion $succ(v, \sigma)$.

```
v_σ ← v
for all s ∈ v do
   for all p^t ∈ s do
      if â_{p,t} ∈ σ̄ then
         v_σ ← v_σ \ s
      else if â_{p,t} ∈ (A_F ∩ σ) then
         for all s' ∈ v do
            if parallel(s, p^t, s') then
               v_σ ← v_σ \ s'
            end if
         end for
      end if
   end for
end for
if v_σ = ∅ then
   return ∅
end if
v' ← ∅
for all s ∈ v_σ do
   v' ← v' ∪ succ(s)
end for
return v'
```

In der Praxis ist zu beobachten, dass die Anzahl der Annahmen unabhängig von der Länge der Eigenschaften meist sehr gering ist. Dadurch tritt oben genannter Effekt nur bei einer wachsenden Anzahl von Eigenschaften auf, und sollte nur für Operationseigenschaftssätze problematisch werden, bei denen Eigenschaften automatisch generiert werden.

Die Laufzeit des Gesamtalgorithmus verhält sich analog zur Anzahl der Zustände des Automaten. Die Laufzeit der Nachfolgerfunktion ist hingegen nicht wie die Laufzeit von $succ(s)$ linear bezüglich der Anzahl Verzweigungen und ausgehenden Kanten des Zustands, sondern sie hat einen zusätzlichen quadratischen Faktor durch die Filterung der Pfade für v_σ. Dieser Faktor kommt zustande, da im Kode, der für die Filterung verantwortlich ist, doppelt verschachtelt über die Pfade iteriert werden muss. Im Gegensatz zur oben genannten exponentiellen Zunahme der Übergänge und damit verbundenen Aufrufe von $succ(v, \sigma)$ verläuft diese Entwicklung jedoch sehr moderat.

4.8 Erzeugung des Schaltsignals der Freezevariablen

In Abschnitt 4.5.4 wurde die Anzahl der Speicherplätze für Freezevariablen anhand der Analyse der Eigenschaftsüberlappung auf ein Minimum reduziert. Dazu war es notwendig, die Daten der Speicherplätze zu bestimmten Eigenschaftszeitpunkten "weiterzuschalten". Das entsprechende Schaltsignal stand zu diesem Zeitpunkt jedoch noch nicht bereit. Erst mit der erfolgreichen Konstruktion des Kontrollautomaten ist die Erzeugung des Schaltsignals möglich.

Das Schaltsignal s_i einer Freezevariable f der Eigenschaft p muss genau dann aktiviert werden, wenn die Eigenschaftsposition $p^{time(f)+i}$ auftritt, d.h. wenn einer der gegenwärtig aktiven Zustände des Automaten diese Eigenschaftsposition enthält. Dazu müssen die Zustände (bzw. Pfade) des Automaten, die eine betreffende Eigenschaftsposition enthalten, gesammelt werden. Die Menge S_{p^t} ist durch folgende Formel definiert:

$$S_{p^t} := \left\{ s \in S \mid p^t \in s \right\}.$$

Anstelle des speziellen, auf die Freezevariable zugeschnittenen Schaltsignals s_i soll ein allgemeines Schaltsignal eingeführt werden, welches auch im weiteren Verlauf der Synthese verwendet wird. Dieses Schaltsignal $\psi_{p,t}$ steht zum Freezevariablenschaltsignal in folgender Relation

$$s_i := \psi_{p,time(f)+i}.$$

Um das allgemeine Signal $\psi_{p,t}$ definieren zu können, muss zwischen beiden Automatentypen unterschieden werden. Wenn zu einem gegebenen Zeitpunkt der NEA eine bestimmte Anzahl aktivierter Zustände S_{akt} besitzt (durch den Nichtdeterminismus können mehrere Zustände zugleich aktiv sein), kann das Schaltsignal mit

$$\psi_{p,t} := \left(S_{akt} \cap S_{p^t} \right) \neq \emptyset \quad (\text{für } \mathcal{A}_{NEA})$$

ermittelt werden, d.h. wenn einer der aktiven Zustände zu den gesuchten Zuständen zählt, wird das Schaltsignal $\psi_{p,t}$ aktiviert.

Im DEA gibt es genau einen aktiven Zustand v_{akt}, der allerdings wiederum aus einer Teilmenge der Zustände des NEA besteht. Damit ist das Schaltsignal

$$\psi_{p,t} := \left(v_{akt} \cap S_{p^t} \right) \neq \emptyset \quad (\text{für } \mathcal{A}_{DEA})$$

ähnlich aufgebaut, wie im nichtdeterministischen Fall.

Damit sind alle Vorraussetzungen für eine erfolgreiche und schnelle Abbildung des durch ITL beschriebenen Systems in Hardwarekonstrukte geschaffen. Diese Abbildung soll Thema des nächsten Abschnitts sein.

4.9 Abbildung in Hardware

Um das in den vorhergehenden Abschnitten erzeugte Modell in eine Netzliste für eine bestimmte Zielplattform (z.B. einen ASIC bzw. einen FPGA-Bitstrom) umwandeln zu können, muss es in eine Beschreibungsform überführt werden, die von gebräuchlichen Synthesewerkzeugen verstanden wird. In dieser Arbeit wurde dazu eine VHDL-Beschreibung auf Register-Transfer-Ebene gewählt, wobei die konkrete Sprache durch die Trennung in zwei Phasen (vgl. Abschnitt 4.1) unerheblich ist. Das Ergebnis von Phase 1 ist ein sprachunabhängiges Zwischenformat, in dem Hardwarekonstrukte in ITL-ähnlicher Semantik abgelegt werden können. Ziel dieses Unterkapitels ist es, die bereits erzeugten formalen Modelle in das Zwischenformat zu überführen.

Bei der Abbildung wird rekursiv vorgegangen, d.h. es existiert ein vom Benutzer vorgegebenes Toplevelmodul mit einem Satz von Parametern (*generics*). Dieses Toplevelmodul wird durch Einlesen des **structure**-Abschnitts direkt in die entsprechende Komponente im Zwischenformat überführt. Untermodule werden rekursiv nach dem gleichen Schema abgebildet. Viele der Abbildungsmechanismen sind entweder trivial, oder wurden bereits bei den modellgenerierenden Schritten der vorherigen Abschnitte dargestellt. Einige wenige Punkte bedürfen jedoch einer weiteren Untersuchung. Zum einen betrifft dies die Abbildung von Ausdrücken und Makros. Zum anderen ist die Hardwaregenerierung der beiden Automatentypen zu betrachten.

4.9.1 Abbildung des Datenflusses - Ausdrücke und Makros

Beispiel

Die Abbildung von Ausdrücken und Makros kann am besten mit Hilfe eines kleinen Beispiels illustriert werden. Dazu ist in Abb. 4.16 ein ITL-Makro abgebildet, das *size* Bits des Signals *data* zusammenfügt. Wenn die Länge des Signals nicht ausreicht, werden rekursiv Datenworte des Signals zu vorangegangenen Zeitpunkten erfasst und verwendet. Die Bedingung der ITL-Anweisung **if statically** muss "statisch" zur Synthesezeit ausgewertet werden und je nach Ergebnis wird nur einer der beiden Zweige weiter betrachtet.

Die beiden Blockschaltbilder in Abb. 4.17 zeigen das Syntheseergebnis für das *seq_prev* Makro, wenn das Argument *size* den Wert 32 hat und das Argument *data* einen unbekannten Wert der Länge 16 Bit. Aufgrund der übergebenen Werte sind lediglich zwei verschachtelte Aufrufe des Makros notwendig, da bereits in der zweiten Stufe der nichtrekursive Zweig des Makros gewählt wird. Aus diesem Grund gibt es zwei verschiedene Instanzen des Makros, für jede Rekursionsstufe eine. Signale, deren konstanter Wert

```
macro seq_prev (data : bit_vector; size : numeric)
    : bit_vector :=
  if statically data'length < size then
    seq_prev(prev(data),size-data'length) & data;
  else
    data(size-1 downto 0);
  end if;
end macro;
```

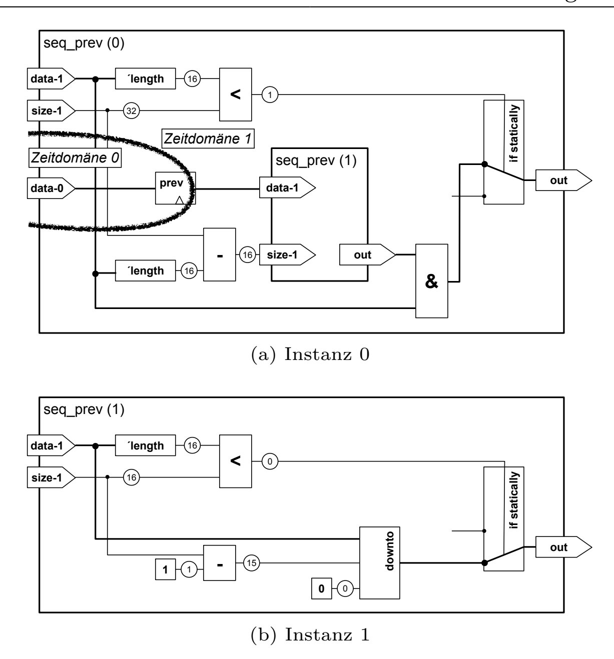
Abbildung 4.16: ITL-Makro zum Zusammenfügen von Datenworten des selben Signals zu unterschiedlichen Zeitpunkten.

zur Synthesezeit bekannt ist, sind dünner gezeichnet und mit dem entsprechenden Wert gekennzeichnet. Der Aufruf des gesamten Makros geschieht zum Zeitpunkt $t + 1$.

Zeitdomänen

Bei der Auswertung und Abbildung von Makros und anderen Ausdrücken ist das Mitführen der sogenannten Zeitdomäne wichtig. Diese drückt aus, zu welchem Zeitpunkt der Ausdruck ausgewertet wird. Die Zeitdomänen richten sich nicht nach Makrogrenzen und verändern sich bei jedem Auftreten eines zeitlichen Operators. Bei **prev** verringert sich der Zeitwert um die entsprechende Anzahl Zeitschritte, während er sich bei einem **next** erhöhen würde, wenn dieser Operator denn implementierbar wäre (vgl. Abschnitt 3.3.3).

Wird ein Argument eines Makros in mehreren Zeitdomänen gleichzeitig genutzt (siehe Argument *data* im Beispiel), muss es auch mehrfach übergeben werden. In Abb. 4.17a existieren deswegen zwei Eingangsports für das Signal *data*. Der Port $data - 0$ stellt das Signal zum Zeitpunkt 0 bereit, während $data - 1$ zum darauf folgenden Zeitpunkt das korrekte Ergebnis liefert. Einziger Grund für die Zeitdomänen und diese mehrfache Übergabe sind die Freezevariablen. Das Signal einer Freezevariablen ist je nach Zeitdomäne verschieden. Die Verwendung einer Freezevariablen f in einer Zeitdomäne vor dem Freezezeitpunkt $time(f)$ ist generell nicht zulässig. Die Zuordnung geschieht anhand der Signale f_i oder $f_{[i,j]}$, die in Abschnitt 4.5.4 entsprechend der beiden alternativen Implementierungsvarianten definiert wurden. In Abb. 4.18 sind zwei verschiedene Aufrufe des Makros *seq_prev* dargestellt. Zum einen kann beim direkten Aufruf mit einem Eingangssignal *insig* das entsprechende Signal unmittelbar mit allen Zeitdomänen verknüpft wer-



(a) Instanz 0

(b) Instanz 1

Abbildung 4.17: Hardwareabbildung des Makros *seq_prev* zur Ausführungszeit 1 mit *data* als 16-Bit Vektor und *size* = 32.

den. Zum anderen muss bei der Verwendung einer Freezevariable beachtet werden, dass die Zeitdomänen mit den passenden Speichersignalen der Freezevariable verknüpft werden. Der Ausgang der beiden Blöcke ist jeweils ein Annahmensignal *assume-1*.

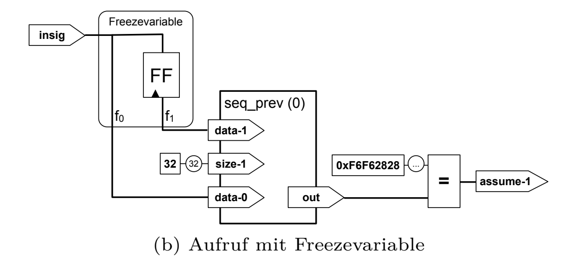
Suche im Signalverzeichnis

In der Implementierung von *vhisyn* werden alle Freezevariablen und Makroargumente zusammen mit ihrer Zeitdomäne im lokalen Signalverzeichnis abgelegt (vgl. Abschnitt 4.3.1). Bei der rekursiven Auswertung von Ausdrücken werden auftretende Bezeichner anhand der aktuellen Zeitdomäne im jeweiligen lokalen Verzeichnis gesucht und im Erfolgsfall mit dem entsprechenden Signal verknüpft. Fall kein Eintrag im lokalen Verzeichnis vorliegt, wird im globalen Verzeichnis nach dem Bezeichner gesucht (nur Eingangssignale und Parameter). Falls der Bezeichner dort gefunden wurde, wird das Signal

```
assume:
   at t+1 :   seq_prev(insig,32) = X"F6F6A2A2";
```

(a) Aufruf mit Inputsignal

```
freeze:
   f = insig @ t;
assume:
   at t+1 :   seq_prev(f,32) = X"F6F6A2A2";
```



(b) Aufruf mit Freezevariable

Abbildung 4.18: Hardwareabbildung des Aufrufs von Makro *seq_prev*.

von der obersten Ebene in die aktuelle Aufrufebene hinein verbunden und genutzt. Dabei kommen auch die lokalen Signalverzeichnisse als Zwischenspeicher zum Einsatz und verhindern, dass ein Signal mehrfach über die verschiedenen Ebenen hinweg verbunden wird.

Verzögerte und bedingte Auswertung

Ein weiterer Punkt, der mit der Berücksichtigung der Freezevariablen zusammenhängt, ist die verzögerte Auswertung von Makroargumenten. Da jede Auswertung in einer bestimmten Zeitdomäne stattfinden muss, und beim Aufruf eines Makros die Zeitdomänen seiner Argumente noch nicht feststе-

hen, muss die Auswertung der Argumente verzögert werden, bis sie im Makrokörper benötigt wird und auch die konkrete Zeitdomäne feststeht.

Um die Anweisung **if statically** korrekt auswerten zu können, dürfen die beiden Verzweigungskörper nicht sofort ausgewertet werden. Es muss zuerst auf das Ergebnis der Verzweigungsbedingung gewartet werden, um dann in Abhängigkeit des logischen Wertes nur die richtige Verzweigung zu wählen.

Algorithmus

Die Auswertung eines Ausdrucks ist eine der elementaren Routinen in *vhisyn*. Sie basiert auf zwei zentralen Stapelspeichern, dem *Ausdruckstapel* und dem *Ergebnisstapel*. Die Einträge des Ausdruckstapels bestehen aus einem Tupel von Ausdruck (in Form des zugehörigen abstrakten Syntaxbaumes (AST)) und der Zeitdomäne des Ausdrucks. Im Gegensatz dazu besteht der Ergebnisstapel aus Tupeln, die sich aus Signalen (Netz-ID) und konkreten Signalwerten zusammensetzen. Der Wert ist im Falle eines konstanten Ausdrucks eben diese Konstante. Falls keine konstante Auswertung möglich ist, wird der Wert "unbekannt" verwendet.

Anfangs wird der gesamte auszuwertende Ausdruck zusammen mit der geforderten initialen Zeitdomäne auf dem Ausdruckstapel abgelegt. Im weiteren Verlauf wird immer der oberste Ausdruck von diesem heruntergenommen und ausgewertet. Dabei werden neu entstehende Teilausdrücke auf den Ausdruckstapel gelegt oder Ergebnisse in Form von Signal-Wert-Paaren vom Ergebnisstapel heruntergenommen bzw. daraufgelegt.

Je nach Art des obersten Ausdrucks finden verschiedene Operationen statt. Im folgenden sollen diese Operationen aufgeführt werden, wobei jeweils nur die wichtigsten Aufgaben beschrieben sind:

- Für *Konstanten* wird die entsprechende Hardwarekonstante angelegt und das entstehende Signal sowie der Wert der Konstante auf den Ergebnisstapel gelegt.

- *Lokale Bezeichner*, die in einem der Signalverzeichnisse gefunden werden, können direkt verwendet werden, d.h. der Eintrag wird auf den Ergebnisstapel gelegt.

- Ist der gefundene Ausdruck ein Bezeichner, der ein *Argument* des aktuellen Makroaufrufs darstellt, wird der Ausdruck im übergeordneten Makroaufruf gesucht, der diesem Argument übergeben wurde. Der Ausdruck wird auf den Ausdruckstapel gelegt. Nach Beendigung dieser Auswertung liegt das Ergebnis auf dem Ergebnisstapel und wird im lokalen Signalverzeichnis zwischengespeichert.

- *Globale Bezeichner* werden wie in Abschnitt 4.3.1 beschrieben behandelt und das Ergebnis wird ebenfalls auf den Ergebnisstapel gelegt.

- *Makroaufrufe* führen zuerst zu einem Speichern des Makrokörpers auf dem Ausdruckstapel. Nach abgeschlossener Auswertung liegt dann das Ergebnis des Makroaufrufs auf dem Ergebnisstapel.

- Bei Anweisungen von **if statically** werden zuerst die beiden Verzweigungsausdrücke auf den Ausdruckstapel gelegt, gefolgt vom Ausdruck der Bedingung. Wenn die Bedingung ausgewertet wurde, wird ihr Ergebnis vom Ergebnisstapel genommen und untersucht. Wenn der Signalwert logisch wahr ergibt, wird der **else**-Teil vom Ausdruckstapel entfernt. Im anderen Fall wird der **then**-Teil verworfen. Dabei muss in einem der beiden Fälle nicht der oberste Wert vom Stapelspeicher entfernt werden sondern der darunter liegende.

- Der zeitliche Operator **prev** legt sein Argument, d.h. den zu verzögernden Ausdruck, mit einer entsprechend angepassten Zeitdomäne auf den Stapel. Wenn der Operator ein zweites Argument besitzt, welches die Anzahl der Verzögerungsschritte angibt, muss zunächst dieses Argument auf den Stapel gelegt und auf die Auswertung der Verzögerung gewartet werden. Im Anschluss wird der Wert der Verzögerung als Konstante vom Ergebnisstapel entfernt und das erste Argument von **prev** wird dem Ausdruckstapel hinzugefügt. Dabei muss die geforderte Zeitdomäne mit Hilfe der Verzögerungskonstante angepasst werden.

- Alle *weiteren Operatoren*, inklusive des nicht-statischen Verzweigungsoperators (normales **if**), legen ihre gesamten Argumente auf den Ausdruckstapel und lesen nach beendeter Auswertung die entsprechenden Werte vom Ergebnisstapel. Das Gesamtergebnis der Operation wird errechnet und wiederum auf den Ergebnisstapel gelegt. Weiterhin wird auch der zugehörige Hardwareoperator angelegt und mit den passenden Signalen verknüpft. Wenn die Werte der Argumente es zulassen, kann das Ergebnis auch als Konstante weiterverarbeitet werden.

Der Algorithmus ist beendet, wenn der Ausdruckstapel leer ist. In diesem Fall befindet sich nur ein einziger Eintrag auf dem Ergebnisstapel. Er stellt das Ergebnis des gesamten Ausdrucks dar. Wenn das Ergebnis des Ausdrucks nur als Konstante benötigt wurde, kann der Wert des Ergebnisses direkt verwendet werden. Dies tritt zum Beispiel bei der Auswertung der **for timepoints** Sektion einer Eigenschaft oder bei der Übergabe von Werten für Parameter von hierarchischen Unterkomponenten auf. Falls das Ergebnis in

einem dynamischen Kontext, wie einer Annahme oder dem Zuweisungsteil einer Zusicherung auftritt, wird nur die Hardwareentsprechung des Signals in Form der Netz-ID verwendet.

Rekursion

Die beschriebene Aufgabenstellung kann anstatt mit einer iterativen Lösung unter Nutzung von Stapelspeichern auch mit einem rekursiven Algorithmus bearbeitet werden. Die Rekursion des Auswertungsalgorithmus würde dann den verschachtelten Aufrufen der Ausdrücke folgen. In der Praxis bedeutet dies aber sehr hohe Rekursionstiefen, die wiederum an die Grenze der maximalen Rekursionstiefe der Implementierung stossen. Im Allgemeinen hat sich die iterative Implementierung als robustes und flexibles Verfahren erwiesen, welches auch bei einer sehr hohen Tiefe des Aufrufbaumes von mehreren tausend Ebenen uneingeschränkt nutzbar ist.

4.9.2 Direkte Abbildung des nichtdeterministischen Automaten

Synthesefähige Hardware verhält sich prinzipiell deterministisch. Der aktuelle Zustand wird dabei in Speicherelementen (Registern) gehalten. Mit Hilfe einer rein kombinatorischen Zustandsübergangsfunktion kann der Folgezustand bestimmt werden, der dann den vorherigen Wert in den Zustandsregistern ablöst.

Verschiedene Automatenkodierungen

Die Kodierung der Zustände S in den Registern ist ein wichtiges Implementierungsmerkmal eines Automaten. Um einen Überblick zu gewinnen und das folgende Verfahren besser einordnen zu können, sollen in der folgenden Aufzählung die drei gebräuchlichsten Verfahren vorgestellt werden.

- Die flächeneffizienteste Kodierung ist die *Binärkodierung* bei der exakt $\lceil \log_2 |S| \rceil$ Register benötigt werden. Jede gültige Belegung der Register entspricht einem Zustand. Der wesentliche Vorteil ist der geringe Verbrauch an Registern, während in einigen Fällen die Zustandsübergangsfunktion aufwendiger wird.

- Eine spezielle Variante der Binärkodierung ist die *Gray-Kodierung*, die zur Abbildung von Zuständen auf Registerbelegungen den Gray-Kode verwendet. Damit wird erreicht, dass bei Übergängen zwischen benachbarten Zuständen nur eines der Register seinen Wert ändern muss.

- Bei der sogenannten *One-Hot-Kodierung* hingegen existieren $|S|$ Register, sodass nur ein Teil der möglichen Belegungen genutzt wird. Jede gültige Belegung enthält genau ein Register mit dem Wert 1, während alle anderen nicht aktiv sind. Weiterhin findet eine eindeutige Zuordnung eines Registers auf einen Zustand statt. Im Gegensatz zur Binärkodierung wird eine höhere Anzahl Register benötigt. Vorteilhaft wirkt sich hingegen die oft einfachere Zustandsübergangsfunktion aus.

Überführung in den Potenzmengenautomat

Um die in Abschnitt 4.6 beschriebenen nichtdeterministischen Automaten in Hardware abbilden zu können, muss eine Transformation in einen DEA durchgeführt werden. Dabei bieten sich zwei Möglichkeiten. Zum einen kann der NEA durch das bereits beschriebene allgemeine Potenzmengenverfahren in einen DEA überführt werden. Der resultierende Potenzmengenautomat hat $2^{|S|}$ Zustände, was bei einer One-Hot-Kodierung zu ebensovielen Registern führen würde und nicht realistisch durchführbar ist.

Aus diesem Grund sollte eine Binärkodierung gewählt werden. Diese hat $\lceil \log_2 2^{|S|} \rceil$ Zustände, was durch Vereinfachung der Formel in $|S|$ Registern resultiert. Dies würde eine effiziente Implementierung ermöglichen. Nachteilig wirkt sich jedoch aus, dass der Potenzmengenautomat vollständig konstruiert werden muss. Die dabei notwendige Enumerierung aller Elemente der Potenzmenge führt zu exponentiellem Ressourcen- und Laufzeitverbrauch.

Ein Verfahren, das die Konstruktion des Potenzmengenautomaten vermeidet, ist erforderlich und wird im Folgenden vorgestellt. Dabei sollte die Ergebnisimplementierung mit $|S|$ Registern auskommen.

Multiple-Hot-Kodierung

Das erstmals von Sidhu und Prasanna [SP01] vorgestellte Verfahren "emuliert" den NEA in Hardware. Es ähnelt der One-Hot-Kodierung eines DEA insofern, dass jeder Zustand einem Register zugeordnet ist. Im Gegensatz dazu können jedoch zu einem bestimmten Zeitpunkt auch mehrere Register gleichzeitig aktiv sein, weswegen wir das Verfahren *Multiple-Hot-Kodierung* nennen. Die Einsatzgebiete umfassen unter anderem die Synthese von Automaten zur Erkennung von regulären Ausdrücken [SP01], sowie die Abbildung von nichtdeterministischen Automaten in der High-Level-Synthese von Protokollspezifikationen [Sie05].

In Abb. 4.19 wird die Multiple-Hot-Kodierung für den bereits in Abb. 4.9a dargestellten NEA gezeigt. Man kann erkennen, dass das Verfahren für jeden Zustand s exakt ein Register $R(s)$ erzeugt. Die Zustandsübergangsfunktion

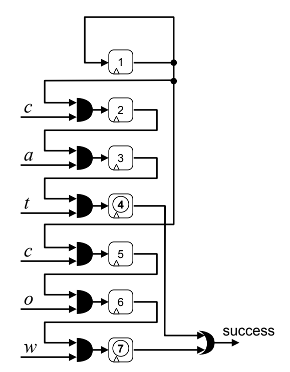


Abbildung 4.19: Multiple-Hot-Kodierung des NEA zur Erkennung der Zeichenketten "cat" oder "cow" im Eingabestrom.

muss für jedes Register einzeln errechnet werden. Der nächste Wert $R'(s)$ ergibt sich zu

$$R'(s) := \bigvee_{(u,\sigma)}^{\{(u,\sigma)|(u,\sigma,s)\in\delta\}} R(u) \wedge \sigma.$$

Dabei wird für jede Transition, die den Zustand s zum Ziel hat, das Register des Vorgängerzustands u mit der Übergangsbedingung σ konjunktiv verknüpft. Diese Verknüpfungen werden dann wiederum disjunktiv miteinander verschalten.

Durch dieses Verfahren kann ein beliebiger NEA direkt in eine entsprechende Hardwarerepräsentation überführt werden, die den NEA "emuliert". Es werden dabei genauso viele Register benötigt, wie der NEA Zustände hat. Die Implementierung kann als geschicktes Verfahren zur direkten Abbildung der Binärkodierung des Potenzmengenautomaten verstanden werden, bei der der Potenzmengenautomat nicht konstruiert werden muss.

Das lineare Wachstum der Anzahl Register mit der Anzahl Zustände, ist jedoch in vielen Fällen ungünstig. So können Eigenschaftssätze mit sehr langen Eigenschaften (bspw. 1000 Takte oder mehr) in einer sehr großen Anzahl Register resultieren. Die entstehende Hardware kann von einem modernen Synthesetool nur noch schwer erfasst und vor allem optimiert werden. Das war am Anfang der Entwicklung von *vhisyn* der Hauptgrund für die Suche nach dem Verfahren zur Generierung des deterministischen Automaten (vgl. Abschnitt 4.7).

4.9.3 Abbildung des deterministischen Automaten

Im Gegensatz zum nichtdeterministischen Automaten ist die Abbildung des DEA sehr einfach möglich. Es wird natürlicherweise eine Binärkodierung verwendet, da im Falle einer One-Hot-Kodierung aufgrund des höheren Registerverbrauchs die Vorteile der Konstruktion des DEA entsprechend Abschnitt 4.7 nicht zum Tragen kommen.

Eine weitere Frage bleibt jedoch offen. Es muss die exakte Zuordnung von Registerbelegungen zu Zuständen gefunden werden. Dabei ist es hilfreich die Registerbelegung als numerischen Wert zu betrachten. Dieser numerische Wert der Registerbelegung eines Zustands $v \in V$ wird im Folgenden als

$$num(v)$$

bezeichnet. Die numerische Betrachtung ermöglicht wiederum einige Optimierungen, die insbesondere bei langen Eigenschaften eine deutlich effizientere Hardware zur Folge haben.

Zustandsketten

Es kann in vielen Fällen beobachtet werden, dass bei langen Eigenschaften viele Zustände eine lange Kette bilden. Dabei sind insbesondere diejenigen Zustände interessant, die genau eine ausgehende Transition ohne Übergangsbedingung besitzen.

In Abb. 4.15 bilden bspw. die Zustände

$$\left\{\{inframe^3\}\right\}$$

bis

$$\left\{\begin{Bmatrix} inframe^9 \\ inframe^1 \end{Bmatrix}, \begin{Bmatrix} inframe^9 \\ nosync^1 \end{Bmatrix}\right\}$$

eine solche Kette. Mit zunehmender Eigenschaftslänge wächst die Anzahl der Zustände in solchen Ketten, während die Anzahl der restlichen Zustände konstant bleibt. Lediglich mit steigender Überlappung und Parallelität der Eigenschaften sinkt die Anzahl der Kettenzustände.

Inkrementelle Zustandsübergänge

Um den Ressourcenverbrauch des DEA gering zu halten, kann darauf geachtet werden, dass der numerische Registerwert von möglichst vielen Kettenzuständen des Automaten eine konstante numerische Differenz zu ihrem Nachfolger hat. Die Übergänge zwischen der betroffenene Zuständen werden

dann in der Beschreibung des Automaten nicht einzeln implementiert sondern können über einen "Defaultübergang" abgedeckt werden. Dieser spezielle Übergang greift immer dann, wenn keiner der anderen Übergänge aktiviert wurde. Er besteht lediglich aus einem Addierer, der einen konstanten numerischen Wert zum gegenwärtigen Zustand addiert, um den neuen Zustandswert zu bestimmen.

Die genaue Abbildung von Zustand auf numerischen Registerwert stellt vor diesem Hintergrund ein Optimierungsproblem dar. Es muss eine Abbildung gefunden werden, die die Anzahl der Zustände, auf die ein Defaultübergang zutrifft, maximiert. Auf eine exakte Definition des Optimierungsproblems soll verzichtet werden, da im Folgenden eine heuristische Lösung vorgestellt wird, die hinreichend gut funktioniert. Das bedeutet, dass mit steigender Eigenschaftslänge und gleichbleibender Überlappung und Parallelität, die Anzahl der nicht vom Defaultübergang betroffenen Zustände konstant bleibt.

Der übliche Wert der Differenzkonstante ist 1, d.h. der numerische Wert des Nachfolgers eines Kettenzustands ist gleichzeitig der Nachfolger des numerischen Vorgängerwertes in den natürlichen Zahlen. Der Wert der Differenzkonstante ist allerdigns unerheblich für das Finden der optimalen Lösung, weil die Folge von numerischen Werten beliebig "umsortiert" werden könnte, sodass um die Differenzkonstante verschiedene Werte direkt nebeneinander liegen.

Heuristik

Als einfachste und zugleich sehr effiziente Heuristik für die Optimierung der Defaultübergänge bietet sich eine lexikographische Sortierung der Zustände des DEA an. Eine lexikographische Ordnung entspricht der Sortierung der Wörter in einem Lexikon, bei der zuerst der erste Buchstabe verglichen wird und nur bei Gleichheit der nächste Buchstabe herangezogen wird. Die lexikographische Ordnungsrelation wird im Folgenden als $<^d$ angegeben.

Die Ordnung ähnelt der partiellen Ordnung von Eigenschaftspositionen in Abschnitt 4.7.5. Für die Sortierung der Zustände reicht jedoch keine partielle Ordnung, sondern es muss eine lineare Ordnung eingeführt werden. Dabei wird zuerst nach dem Eigenschaftsnamen sortiert und bei Gleichheit nach dem Zeitpunkt der Eigenschaft

$$p^t < q^u := (p < q) \vee \left(\overline{(q < p)} \wedge (t < u)\right).$$

Die Sortierung der Eigenschaften ist dabei beliebig und wurde in dieser Arbeit entspechend der alphabetischen Sortierung des Namens gewählt.

Diese Ordnung kann auf nichtdeterministische Zustände (Pfade) $s \in S$ erweitert werden, indem die Eigenschaftspositionen innerhalb eines Pfades

sortiert und nachfolgend zwei Pfade lexikographisch miteinander verglichen werden

$$s_1 < s_2 := (s_1, <) <^d (s_2, <).$$

Zwei Zustände $v_1, v_2 \in V$ des DEA werden analog sortiert

$$v_1 < v_2 := (v_1, <) <^d (v_2, <).$$

Mit Hilfe dieser Ordnung werden die Zustände sortiert und anschliessend durchnummeriert. So erhalten Zustände mit gleichen Eigenschaften, aber benachbarten Zeitpunkten, aufeinanderfolgende numerische Werte.

Beispiel

Um die Abbildung der Zustände auf numerische Registerwerte und den Ressourcengewinn mit Hilfe des Defaultübergangs illustrieren zu können, wurden die Zustände des DEA aus Abb. 4.15 auf numerische Werte abgebildet. In Abb. 4.20 sind die Zustände entsprechend mit diesen Werten beschriftet. Dabei kommt die lexikographische Sortierung der Zustände zum Einsatz.

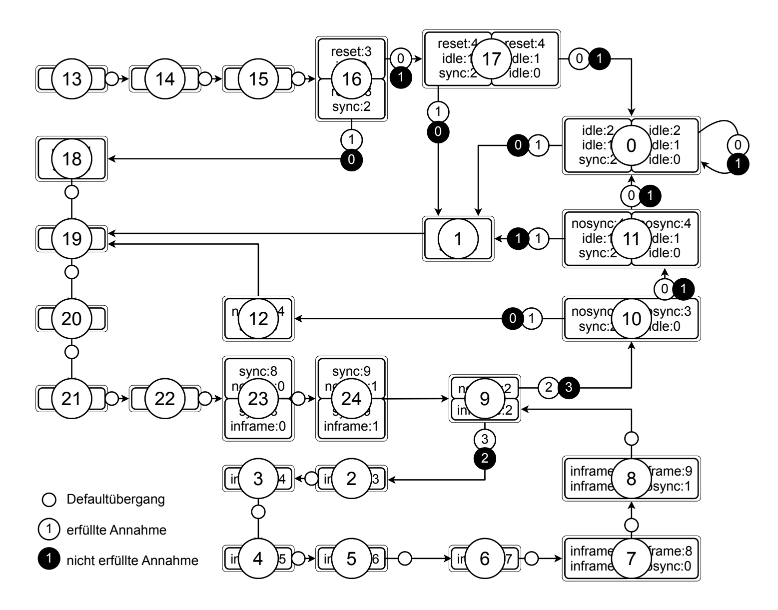


Abbildung 4.20: Abbildung der Zustände des DEA aus Abb. 4.15 auf numerische Werte, entsprechend der lexikographischen Sortierung.

Es können deutlich die Kettenzustände abgelesen werden. Sie sind etwas heller dargestellt und umfassen Zustände der *reset* (13 bis 15), *sync* (1, 12 und 18 bis 24) und *inframe* (2 bis 8) Eigenschaften. In diesen 19 Kettenzuständen befinden sich 16 Zustände, deren Nachfolger ein konstantes Inkrement aufweist. Diese Übergänge sind wiederum kreisförmig markiert. Der Defaultübergang findet in 16 Fällen statt, gegenüber 9 normalen Transitionen.

Da die entstehende Automatenimplementierung als Blockschaltbild nur aufwendig darzustellen ist, zeigt Abb. 4.21 den VHDL-Quellkode des Prozesses, der den DEA implementiert. Der auf das Wesentliche reduzierte Prozess, enthält nur 15 explizit beschriebene Übergänge, während alle weiteren Transitionen durch den Defaultübergang in Zeile 28 abgedeckt sind.

4.9.4 Zuweisung der Zusicherungen

In diesem Abschnitt soll die Abbildung der Ausgaberelation ein wenig genauer beleuchtet werden. Die Definition in Abschnitt 4.6.3 ordnet jedem Zustand eine Teilmenge aller Zusicherungen zu. Bei der Abbildung in Hardware erweist sich jedoch der umgekehrte Fall als günstig, indem den Zusicherungen eine Menge von Zuständen zugeordnet wird, in denen die jeweilige Zusicherung ausgeführt werden soll. Im Weiteren wird ein Schaltsignal benötigt, welches für eine bestimmte Zusicherung (bspw. $z_{p,n,t}$) die Aktivierungsbedingung darstellt. Das in Abschnitt 4.8 definierte Signal $\psi_{p,t}$ erfüllt genau diese Funktion.

Multiplexer-Kaskade

Damit kann für jedes Ausgangssignal oder interne Signal die entsprechende Hardware generiert werden. In Abb. 4.22 ist ein Teil einer solchen Struktur dargestellt. Es kommt eine Kaskade von Multiplexern zum Einsatz, von denen jeder eine der Zusicherungen für das Signal bearbeitet. Als jeweiliges Schaltsignal wird eben erwähntes $\psi_{p,t}$ genutzt, während die Dateneingänge den eigentlichen Zuweisungen entsprechen. Die in der Abbildung genutzte Bezeichnung **n @ p:t** steht dabei für das Signal der Zuweisung $z_{p,n,t}$. Im gezeigten Beispiel bestehen die Zuweisungen lediglich aus Konstanten. In komplexeren Zuweisungen können an dieser Stelle aufwendige kombinatorische Blöcke stehen.

Die Reihenfolge der Multiplexer ist funktional unwichtig, da vom Entwerfer sichergestellt werden muss, dass zu jedem Zeitpunkt nur Zusicherungen gleichzeitig aktiviert werden, die sich nicht widersprechen, d.h. den gleichen Signalwert zuweisen. Der Entwerfer kann dies garantieren, indem er durch

```vhdl
process(clk,rst,reset)
begin
  if rst = '1' then
    state <= 13;
  elsif clk'event and clk = '1' then
    case state is
      when  0 => if assume_0 then state <=  0;
                             else state <=  1;
                 end if;
      when  1 => state <= 19;
      when  9 => if assume_2 then state <= 10;
                             else state <=  2;
                 end if;
      when 10 => if assume_0 then state <= 11;
                             else state <= 12;
                 end if;
      when 11 => if assume_0 then state <=  0;
                             else state <=  1;
                 end if;
      when 12 => state <= 19;
      when 16 => if assume_0 then state <= 17;
                             else state <= 18;
                 end if;
      when 17 => if assume_0 then state <=  0;
                             else state <=  1;
                 end if;
      when 24 => state <=  9;
      when others => state <= state + 1;
    end case;
  end if;
end process;
```

Abbildung 4.21: VHDL-Implementierung des Kontrollautomaten im *Framer* Beispiel.

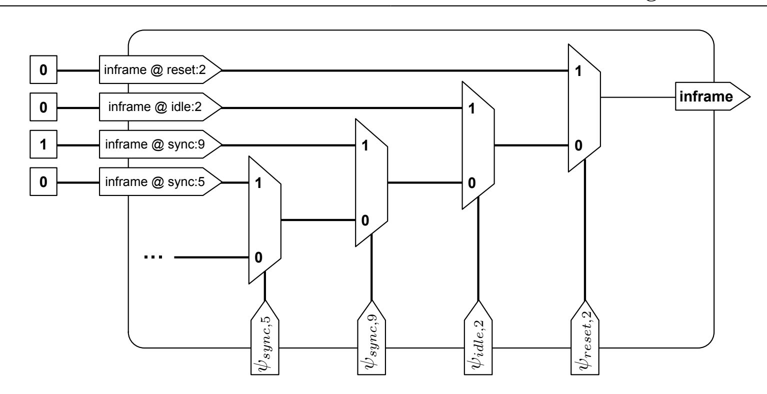


Abbildung 4.22: Teil der Kaskade der Zusicherungen für das Ausgangssignal $inframe$ im *Framer*-Beispiel.

die Vollständigkeitstests die Konsistenz der Eigenschaften prüft und durch einen formalen Nachweis der Eigenschaften auf dem erzeugten Entwurf die Realisierbarkeit ermittelt.

Mehrfach-Multiplexer

Die Performance der Hardware kann jedoch durch die Reihenfolge beeinflusst werden, da das nachgelagerte Logiksynthesetool die gesamte Kaskade je nach Reihenfolge unterschiedlich optimiert. Wenn die Reihenfolge ungünstig gewählt wurde, können unter Umständen Optimierungsmöglichkeiten nicht erkannt werden. Einen Beitrag zur Umgehung des Problems bietet die Ersetzung der Multiplexer-Kaskade durch einen einzelnen Mehrfachmultiplexer, der mit Hilfe von n Auswahlsignalen einen der n Dateneingänge durchschaltet. Welcher Eingang gewählt wird, wenn mehrere Auswahlsignale aktiviert sind, ist dann ein Freiheitsgrad des Multiplexers.

Belegung des Alternativwertes

Bei beiden Implementierungen, sowohl der Multiplexer-Kaskade als auch des Mehrfachmultiplexers, muss der Fall beachtet werden, dass keines der Auswahlsignale aktiv ist. Bei erfolgreichem Determinierungstest kann diese Situation bei internen Signalen auftreten, da diese nur zu denjenigen Zeitpunkten bestimmt sein müssen, zu denen sie auch genutzt werden. Weiterhin kann auch eine bedingte Determiniertheit von Ausgangssignalen vorliegen, sodass

nicht immer eine der Zuweisungen aktiv ist. Es gibt drei Möglichkeiten das Problem zu behandeln und den Alternativwert festzulegen.

Eine Lösung ist das Anlegen eines "don't care" Wertes, der die Unbestimmtheit ausdrückt. Dieses Vorgehen gibt der Logiksynthese weiteres Optimierungspotenzial. In einer Simulation des generierten Designs ist bei dieser Lösung auch im Waveform Fenster ersichtlich, dass im Moment kein gültiger Wert an diesem Signal anliegt. Diese "visuelle" Debugmöglichkeit ist vorteilhaft für das schnelle Finden eines möglichen Fehlers in den Zusicherungsdefinitionen. Nachteilig wirkt sich aus, dass bspw. im Rahmen der Prüfung mit OneSpin 360MV ein Test durchgeführt wird, der die Verwendung von "don't care" Werten stark einschränkt.

Alternativ kann auch die erste Stufe der Kaskade weggelassen werden und deren Zuweisung als Alternativwert genutzt werden. Dies spart etwas Hardwareaufwand und resultiert in einem vorhersehbaren Verhalten nach der Logiksynthese.

Die letzte Möglichkeit ist die Zuweisung einer Konstante als Alternativwert. Dafür bietet sich bspw. der Nullvektor an, der dann bei der Fehlersuche einfach erkannt wird.

Äquivalente Zusicherungen

In Abschnitt 4.5.6 wurden Bedingungen definiert, die es erlauben zwei Zusicherungen z_1 und z_2 auf Äquivalenz zu überprüfen. Falls beide gleich sind ($z_1 \sim z_2$), werden auch ihre entsprechenden Signale, bzw. Netz-IDs gleich sein $expr(z_1) = expr(z_2)$.

Diese Äquivalenz kann genutzt werden, um die Zuweisungshardware zu vereinfachen. Für jedes Ausgangs- oder interne Signal n werden die Äquivalenzklassen $[z]$ aller betroffenen Zusicherungen ermittelt

$$Z(n)/\sim := \{[z] | z \in Z, target(z) = n\}.$$

Weiterhin wird für jede dieser Äquivalenzklassen ein gemeinsames Schaltsignal erstellt, welches sich aus den Schaltsignalen der einzelnen Zusicherungen disjunktiv zusammensetzt

$$\psi_{[z]} := \bigvee_{z_{p,n,t}}^{[z]} \psi_{p,t}.$$

Nun muss entsprechend der Implementierungsmöglichkeiten der vorhergehenden Abschnitte eine Multiplexerkaskade oder ein großer Mehrfachmultiplexer konstruiert werden, der bei Auftreten eines dieser Schaltsignale das

Signal der passenden Zusicherungsklasse auf den Ausgang durchschaltet. Insbesondere bei langen Eigenschaften verspricht dieser Ansatz eine beträchtliche Reduzierung der Multiplexerstufen, bzw. -eingänge.

Optimierung der Schaltsignale des DEA

Ein weiterer Ansatz Hardwareressourcen zu schonen und eine übersichtlichere generierte Beschreibung zu erzeugen, ist die Optimierung der Disjunktion $\psi_{[z]}$ der einzelnen Schaltsignale. Anstatt jedes Schaltsignal einzeln zu verknüpfen, kann der numerische Wert des Automatenzustands herangezogen werden, um zu prüfen ob er in einem Intervall von Schaltsignalen liegt.

Das Schaltsignal $\psi_{[z]}$ muss immer dann gesetzt werden, wenn eine Eigenschaftsposition p^t im aktuellen Zustand $v_{akt} \in V$ auf eine der Zusicherungen $z_{p,n,t} \in [z]$ zutrifft. Dazu werden in einem ersten Schritt ähnlich Abschnitt 4.8 diejenigen Zustände gesammelt, die eine der Eigenschaftspositionen enthalten

$$V_{[z]} := \{v \in V \mid \exists_s^v p^t \in s\}.$$

Nun kann $\psi_{[z]}$ trivialerweise mit

$$\psi_{[z]} := v_{akt} \in V_{[z]} \tag{4.1}$$

definiert werden, was die Voraussetzung für die im Folgenden vorgeschlagene Optimierung bildet.

Wenn für die Zustände $v \in V_{[z]}$ jeweils nur der numerische Wert $num(v)$ betrachtet wird, entsteht eine Folge von natürlichen Zahlen

$$N_{[z]} := \{num(v) \mid v \in V_{[z]}\}.$$

Die Gleichung 4.1 kann mit Hilfe von $n_{akt} = num(v_{akt})$ zu

$$\psi_{[z]} := n_{akt} \in N_{[z]}$$

umformuliert werden. Diese Prüfung kann durch eine Aufteilung in Intervalle vereinfacht werden.

An einem Beispiel ist dies schnell illustriert. Es sei eine Folge $N_{[z]} = 1, 2, 3, 4, 7$ gegeben. Die Prüfung kann nun entweder durch

$$(n_{akt} = 1) \vee (n_{akt} = 2) \vee (n_{akt} = 3) \vee (n_{akt} = 4) \vee (n_{akt} = 7)$$

erfolgen oder durch die viel einfachere Intervallprüfung

$$((n_{akt} \geq 1) \wedge (n_{akt} \leq 4)) \vee (n_{akt} = 7).$$

Da die dabei verwendeten Vergleichsoperatoren zwischen den einzelnen Schaltsignalen für jede Zusicherungsklasse wiederverwendet werden können, entsteht an dieser Stelle ein komplexes Optimierungsproblem, welches zusätzlich durch die in Abschnitt 4.9.3 genutze Heuristik beeinflusst wird.

In *vhisyn* wurde ein simpler heuristischer Ansatz gewählt, indem Wiederverwendung zwischen Schaltsignalen ausgeschlossen wird und alle Intervalle mit mehr als einem Element wie beschrieben durch zwei Vergleichsoperatoren geprüft werden. Einelementige Intervalle werden einzeln mit dem aktuellen Zustand verglichen. Eine Unterteilung von größeren in kleinere Unterintervalle findet nicht statt. Mit diesem Ansatz erreicht man eine effiziente Zusammenfassung der Vielzahl von Schaltsignalen bei sehr langen Eigenschaften mit vielen äquivalenten Zusicherungen.

Alle weiteren Optimierungsmöglichkeiten der Schaltsignale werden dem Logiksynthesetool überlassen.

5 Anforderungen an eine synthesefähige Eigenschaftsbeschreibung

Nicht jede Eigenschaftsbeschreibung, die die Vollständigkeitsprüfung bestanden hat, kann mit dem in dieser Arbeit vorgestellten Verfahren uneingeschränkt in eine Hardwarebeschreibung überführt werden. In diesem Kapitel sollen die dabei auftretenden Ausnahmen und Bedingungen untersucht werden.

Zum einen betrifft dies Einschränkungen technischer Natur, wie bspw. nicht unterstützte ITL-Anweisungen und Operatoren oder auch Anforderungen an den Aufbau einer Eigenschaft. Diese Einschränkungen müssen eingehalten werden, damit der Synthesealgoritmus in der Lage ist, korrekte Hardware zu erzeugen.

Zum anderen gibt es auch Anforderungen an den Beschreibungsstil, die zwar einer erfolgreichen Synthese nicht im Weg stehen, jedoch notwendig sind, um die Vorteile der Eigenschaftsbeschreibung voll zur Geltung zu bringen.

Dieses Kapitel stellt die existierenden Probleme vor und versucht mögliche Lösungswege zu skizzieren. Detaillierte Untersuchungen und konkrete Implementierungsarbeiten sind Gegenstand zukünftiger Entwicklungen.

5.1 Technische Einschränkungen

5.1.1 Intervalle in Zeitvariablen

ITL bietet die Möglichkeit Zeitvariablen nicht nur einen festen Wert zuzuweisen, sondern sie über ein Intervall iterieren zu lassen. So kann bspw. mit der Anweisung

```
for timepoints:
  t_start = t + 1,
  t_end   = t_start + 3..5;
```

eine Zeitvariable *t_end* definiert werden, die einen Wert zwischen 4 und 6 annehmen kann. In der Praxis kann man sich eine Eigenschaft, die ein solches Konstrukt enthält, als "zusammengefasste" Version von eigentlich drei

Eigenschaften vorstellen. Jede dieser drei Eigenschaften ist dann für einen der möglichen Zeitpunkte verantwortlich und hat die Zeitvariable *t_end* auf einen festen Wert 4, 5 oder 6 gesetzt.

Die Eigenschaft muss sicherstellen, dass nicht mehrere der so entstehenden Varianten gleichzeitig aktiviert werden können. Oft wird dies durch eine Annahme erreicht, die zum betreffenden Zeitpunkt (*t_end*) erfüllt sein muss. Nur in einer der Varianten ist die Annahme dann tatsächlich erfüllt, sodass ein eindeutiges Verhalten erreicht wird.

Eine konkrete Implementierung des Verifikationswerkzeugs kann bei der Prüfung der Eigenschaft die Varianten als getrennte Eigenschaften betrachten. Dabei kann jedoch der Prüfaufwand erheblich ansteigen. Aufgrund der Ähnlichkeit der Varianten ist es möglich, durch geschickte Optimierung den Prüfaufwand zu reduzieren.

Eine Synthese von Eigenschaften mit Zeitintervallen ist möglich, jedoch in *vhisyn* zur Zeit nicht implementiert. Um dieses Feature zu unterstützen, wäre es wahrscheinlich ausreichend, die betreffenden Eigenschaften analog zur beschriebenen Verifikationsmethodik aufzuspalten und getrennt zu synthetisieren. In einem weiterführenden Ansatz könnten gemeinsam genutzte Teile der Eigenschaften optimiert werden. Genauere Untersuchungen zu den dabei auftretenden Problemen und mögliche Lösungen wurden im Rahmen dieser Arbeit nicht durchgeführt. Es ist jedoch abzusehen, dass dabei die Komplexität des Verfahrens oder der generierten Hardwarebeschreibung nicht maßgeblich ansteigt.

5.1.2 next-Operator

Wie bereits in vorangegangenen Abschnitten angedeutet, kann der Operator **next** nicht direkt in Hardware abgebildet werden. Während der **prev**-Operator den Wert eines Ausdrucks zum vorherigen Zeitpunkt bestimmt und somit nur ein einfaches Verzögerungsgatter (Register) darstellt, existiert kein Gatter, um den zukünftigen Wert eines Ausdrucks "vorherzusagen".

Aus diesem Grund ist der **next**-Operator in *vhisyn* grundsätzlich nicht implementiert. Jedoch kann es zwei Situationen geben, in denen dieser Operator nicht wirklich benötigt wird, bzw. nachträglich "korrigiert" werden kann. Im folgenden sollen beide Fälle näher erläutert werden.

next und prev heben sich auf

Um die Aufhebung der beiden gegensätzlichen Operatoren zu implementieren sind zwei Beobachtungen notwendig. Zum einen kann der Ausdruck

```
a = next(prev(e))
```

zu

$$a = e$$

umgeformt werden und der **next**-Operator ist eliminiert. Ebenso kann im umgekehrten Fall verfahren werden.

Die zweite Beobachtung betrifft das Verschieben der zeitlichen Operatoren innerhalb eines kombinatorischen Ausdrucks. So kann der Ausdruck

$$a = \mathbf{next}(\mathit{Funktion}(e1, e2))$$

als

$$a = \mathit{Funktion}(\mathbf{next}(e1), \mathbf{next}(e2))$$

geschrieben werden. Auch hier kann die Verschiebung in beide Richtungen durchgeführt werden. Lediglich bei der Verwendung von Freezevariablen muss beachtet werden, dass das Verschieben der zeitlichen Operatoren in einen Freezeausdruck hinein den Zugriffszeitpunkt auf die Freezevariable verändert. Der Ausdruck

$$a = \mathbf{next}(\mathit{Funktion}(e1, f_2))$$

bei dem die Freezevariable f zum Zeitpunkt 2 benutzt wird, kann dementsprechend zu

$$a = \mathit{Funktion}(\mathbf{next}(e1), f_1)$$

umgewandelt werden, wobei im ITL-Quelltext anstelle f_1 und f_2 nur f auftritt. Der jeweilige Index wurde nur der Verständlichkeit halber hinzugefügt.

Mit Hilfe dieser beiden Beobachtungen, ist es möglich in manchen Fällen **next**-Operatoren zu vermeiden. So können alle zeitlichen Operatoren in Richtung des Ausgangs des Gesamtausdrucks verschoben werden. Am Ausgang treffen dann **next** und **prev** aufeinander und heben sich im Erfolgsfall auf.

Bei Verwendung dieses Verfahrens würden sich jedoch die rein kombinatorischen Blöcke und somit unter Umständen der kritische Pfad des Designs verlängern, sodass zeitliche Constraints nicht eingehalten werden können. Aus diesem Grund ist es vorteilhaft, für jeden **next**-Operator in beide Richtungen zu suchen, ob entsprechend gegensätzliche **prev**-Operatoren im Ausdruck vorkommen, mit denen er sich aufheben würde. Nur die dabei beteiligten Operatoren werden dann tatsächlich verschoben, während alle anderen ihren Platz nicht verlassen.

next in Annahmen

Wenn ein **next** nicht mit passenden **prev**-Operatoren ausgeglichen werden kann, bleibt noch eine weitere Möglichkeit, es aufzuheben. So kann eine Annahme jederzeit zeitlich nach hinten verschoben werden. Der dabei entstehende **prev**-Operator kann dann zum Auflösen des **next** verwendet werden. Im folgenden konkreten Beispiel sind alle drei Zeilen äquivalent:

```
at t+3 : Funktion(next(e1),next(e2));
at t+4 : prev(Funktion(next(e1),next(e2)));
at t+4 : Funktion(e1,e2);
```

Allerdings kann dieses Verfahren nicht immer angewandt werden, da sich durch die Verschiebung der Annahme andere Einschränkungen ergeben können. Wenn bspw. durch diese Annahme sich widersprechende Zusicherungen zweier Eigenschaften im nächsten Zeitpunkt aufgelöst werden, verursacht die Verschiebung der Annahme einen Konflikt der Zuweisungen. Diese weiteren Einschränkungen werden in den folgenden Abschnitten näher untersucht.

next in Zusicherungen

Im Gegensatz zu Annahmen, können Zusicherungen auf keinen Fall zeitlich verschoben werden. Aus diesem Grund sind **next**-Operatoren in Zusicherungen nicht durch Verschieben der gesamten Zusicherung auflösbar.

In der Praxis treten Fälle auf, in denen eine Verschiebung aller Ausgänge der Komponente möglich ist, was die Latenz des Systems erhöhen würde. Diese gleichzeitige Verzögerung aller Ausgänge lässt **prev**-Operatoren entstehen, die wiederum zur Aufhebung von **next** verwendet werden können. Ohne zusätzliche Informationen über das gewünschte Ein- und Ausgangsverhalten ist ein solcher Schritt jedoch nicht automatisch durchzuführen.

Fazit

Zusammenfassend kann gesagt werden, dass **next**-Operatoren immer dann nicht zu implementieren sind, wenn sie "vorhersagendes" Verhalten erfordern, d.h. das System benötigt Informationen über Eingangssignale zu einem Zeitpunkt in der Zukunft. Alle anderen Fälle, in denen der Operator zwar benutzt wird, aber kein zukünftiges Verhalten vorhergesagt werden muss, können nach den oben beschriebenen Verfahren aufgelöst werden. Da die Notwendigkeit solcher Anwendungen von **next** jedoch sehr beschränkt ist, wird der Operator in *vhisyn* komplett zurückgewiesen.

5.1.3 Weitere nicht unterstützte ITL-Konstrukte

Neben dem **next**-Operator gibt es eine Anzahl weiterer von *vhisyn* nicht unterstützter Konstrukte. Dazu gehören einige der selten benutzten Operatoren, die hauptsächlich dem Nutzerkomfort dienen und bei der Erstellung einer Eigenschaftsbeschreibung durch andere Konstrukte ersetzt oder mit Hilfe von nutzerdefinierten Makros nachgebildet werden können. Da eine genaue Auflistung aller nicht unterstützten Konstrukte stark von der verwendeten ITL-Version abhängt, wird an dieser Stelle darauf verzichtet und es sollen nur der Operator **let** und die nicht unterstützten zeitlichen Anweisungen näher beschrieben werden.

Der let-Operator

Da Makros in ITL in einem Stil beschrieben werden, der der funktionalen Programmierung ähnelt, liegt es nahe, den Operator **let** einzusetzen. Er dient der Definition von Zwischenwerten und ist ein wichtiger Bestandteil von funktionalen Sprachen wie LISP [Ste90] und Haskell [Hud+92]. Als Beispiel soll ein Makro dienen, welches das Quadrat einer Differenz zwischen zwei Zahlen a und b ausrechnet.

```
macro square_diff(a,b : numeric) : numeric :=
  (a-b) * (a-b);
end macro;
```

Der Subtraktionsoperator taucht hierbei in jedem Faktor einmal auf. Dies bedeutet zum einen mehr Schreibaufwand, zum anderen auch einen erhöhten Bedarf an Hardwareressourcen, sofern keine Optimierung im Logiksyntheseschritt stattfindet. In der Regel ist es demnach wünschenswert einen Zwischenwert für die Differenz zu berechnen und diesen zu quadrieren. Mit Hilfe des **let**-Operators ist dies möglich, wie das folgende Makro zeigt.

```
macro square_diff(a,b : numeric) : numeric :=
  let t = a-b in
    t * t;
  end let;
end macro;
```

In der gegenwärtigen Version von *vhisyn* ist dieser Operator nicht implementiert, da das gleiche Ergebnis auch erzielt werden kann, indem ein zweites Makro definiert wird, welches lediglich dem Zweck dient, in seinen Argumenten das Zwischenergebnis abzuspeichern.

```
macro square(t : numeric) : numeric :=
```

```
    t * t;
end macro;

macro square_diff(a,b : numeric) : numeric :=
    square(a-b);
end macro;
```

Zeitliche Ausdrücke

Zeitliche Ausdrücke bzw. Anweisungen können jeweils im Annahmen- oder Zusicherungsteil einer Eigenschaft auftauchen. Dazu gehören bspw. die beiden Schlüsselwörter **at** und **during**, welche von *vhisyn* unterstützt werden und bereits in vorangegangenen Kapiteln vorgestellt wurden. Im Gegensatz dazu existieren einige Anweisungen, die nicht unterstützt werden und vor allem dazu dienen, verschiedene Verhaltensweisen innerhalb einer Eigenschaft anzubieten. Für detailliertere Beschreibungen dieser Operatoren sei auf das Benutzerhandbuch von OneSpin 360MV verwiesen [OSS09].

- Die Anweisung **either** stellt zwei oder mehr alternative zeitliche Ausdrücke bereit, von denen nur einer erfüllt sein muss. Dies kann genutzt werden, um verschiedene Szenarien von Annahmen zu beschreiben. Eine mögliche Implementierung durch *vhisyn* könnte im einfachsten Fall für jede Alternative eine eigene Eigenschaft generieren. Diese Eigenschaften schliessen sich dann jedoch nicht mehr gegenseitig aus, was jedoch nicht zu Problemen führt, da alle Zusicherungen gleich sind.

  Die Verwendung dieser Anweisung im Zusicherungsteil stellt einen Freiheitsgrad der Implementierung dar, da das System nur die Zusicherungen eines der Auswahlblöcke erfüllen muss. Eine solche Mehrdeutigkeit ist bei der Synthese in konkrete Hardware nicht erwünscht.

- Die Anweisung **exists** repräsentiert ebenso eine Auswahl von mehreren Blöcken mit zeitlichen Ausdrücken, von denen nur einer erfüllt sein muss. Im Unterschied zu **either** wird dabei eine Schleifenvariable genutzt, die über einen fest definierten Bereich iteriert. Die Implementierbarkeit der Anweisung verhält sich analog zu **either**.

- Das Gegenteil stellt die Anweisung **foreach** dar, bei der nicht einer der Blöcke erfüllt sein muss, sondern jeder einzelne. Im Annahmenteil ist die Implementierung der Anweisung durch Abrollen der Schleifenvariable und Erstellen einzelner Annahmen für jeden Durchlauf möglich.

Im Zusicherungsteil ist die Anweisung nur umzusetzen, wenn dadurch keine sich widersprechenden Zusicherungen entstehen, d.h. die Schleifenvariable wird nur bei der Berechnung des Zeitpunktes verwendet, was jedoch ebensogut mit **during** implementiert werden könnte.

Zuweisungen auf Teile von Signalen (z.b. durch Iteration über die Bits eines Ausgangssignals) stellen eine weitere synthetisierbare Verwendungsmöglichkeit der **foreach**-Anweisung dar.

- Analog zur **during**-Anweisung existiert das Schlüsselwort **within**, welches ein Intervall vorgibt, innerhalb dessen ein zeitlicher Ausdruck erfüllt sein muss. Eine Implementierung unterliegt den selben Einschränkungen wie die Anweisungen **either** und **exists**. Demnach ist eine Realisierung im Annahmenteil möglich, während sie im Zusicherungsteil zu Mehrdeutigkeiten führen würde.
- Die Verwendung von *temporalen Makros* ist in *vhisyn* nicht möglich, wird jedoch in zukünftigen Versionen der Software angestrebt, um ein breiteres Einsatzgebiet zu ermöglichen. Es wurde bisher auf eine Implementierung dieses Sprachfeatures verzichtet, da sie lediglich dem Komfort des Nutzers dient.
- In ITL sind die Ausdrücke **true** und **false** als zeitliche Anweisungen möglich. Da sie nur in verschiedenen Verifikationsszenarien und beim Debugging zum Einsatz kommen, wurde auch hier auf eine Implementierung verzichtet.

5.1.4 Zusicherungen

Umwandlung von Zusicherungen in Zuweisungen

In der formalen Verifikation beschreibt eine Eigenschaft einen einzelnen Wahrheitswert, d.h. die Eigenschaft ist erfüllt oder sie ist nicht erfüllt. Dabei ist entweder eine der Annahmen nicht erfüllt oder alle Zusicherungen müssen erfüllt sein. Während bei Annahmen sich dieses binäre Verhalten in der Synthese als Zustandsübergangsbedingung am Kontrollautomaten wiederfindet, ergibt sich bei Zusicherungen eine andere Fragestellung. Es muss aus einer Wahrheitsaussage abgeleitet werden, welchen Signalen welcher Wert zugewiesen werden kann.

Obwohl Zusicherungen in der Form

```
prove:
  at t+2 : out < in;
```

erlaubt sind, wird bei der Vollständigkeitsprüfung der Determinierungstest keine eindeutige Bestimmung des Ausgangssignals *out* feststellen können. Aus diesem Grund sind Zusicherungen meist als direkte "Zuweisungen" implementiert, d.h. das Ausgangssignal wird mit dem Resultat eines Ausdrucks verglichen.

```
prove:
  at t+2 : out = Ausdruck;
```

Es kann jedoch auch Fälle geben, in denen das Ausgangssignal eindeutig determiniert ist, aber trotzdem keine direkte Zuweisung stattfindet. Es können bspw. mehrere Zusicherungen im Zusammenspiel den richtigen Wert ermitteln.

```
prove:
  at t+2 : out < in;
  at t+2 : out > in-2;
```

In Ausnahmefällen kann durch Umformen der Ausdrücke eine direkte Wertzuweisung ausgerechnet werden. Im Allgemeinen ist der erforderliche Aufwand aber sehr groß und schlecht einzuschätzen.

Eine andere Sichtweise lässt nicht eindeutig bestimmte Zuweisungen als Freiheitsgrad in die Synthese einfließen, d.h. die Zusicherung $out < in$ wird bspw. willkürlich durch eine konkrete Zusicherung $out = in - 1$ ersetzt.

In *vhisyn* wird nur die direkte Zuweisung mit = unterstützt. Zwei Gründe sprechen gegen eine Verfolgung der eben beschriebenen Ansätze zur Ermittlung des Zuweisungswertes. Zum einen kann sich der entstehende Berechnungsaufwand von Fall zu Fall stark unterscheiden und zum anderen sind in Hardwarebeschreibungen Freiheitsgrade bzgl. mehrerer Implementierungsvarianten nicht wünschenswert.

Eine davon unabhängige Beobachtung betrifft die Datentypen des Signals und des zugewiesenen Ausdrucks. Wenn sich beide unterscheiden stellt sich die Frage wie die Zuweisung auszusehen hat, wenn der Ausdruck Werte annimmt, die im Wertebereich des Signals nicht verfügbar sind. Das Problem ist schnell gelöst, da der Vollständigkeitstest sicherstellt, dass der Vergleich zwischen Signal und Ausdruck immer erfolgreich ist. Das bedeutet im Umkehrschluss, dass der Ausdruck nie einen Wert annehmen kann, der außerhalb des Wertebereichs des Signals liegt. Es genügt demnach eine korrekte Zuweisung in der Schnittmenge der Wertebereiche sicherzustellen. Wenn der Ausdruck bspw. einen größeren Wertebereich hat, können höherwertige Bits einfach verworfen werden.

Zusicherungen vor der linken Grenze der Eigenschaft

In Abschnitt 4.5.5 wurde erläutert, dass Annahmen, wenn sie vor der linken Grenze (**left_hook**) der Eigenschaft liegen, verzögert werden müssen. Dies war darin begründet, dass die zur Eigenschaft passenden Zustände im Kontrollautomaten erst mit dieser linken Grenze starten, und demnach frühere Annahmen keinen konkreten Zuständen zugeordnet werden können.

Auf Zusicherungen trifft dieser Umstand gleichermaßen zu, jedoch ist eine Verzögerung von Zusicherungen nicht möglich. Das heißt, dass *vhisyn* Zusicherungen vor der linken Grenze nicht unterstützt.

Eine mögliche Lösung des Problems ist die Verschiebung der betroffenen Eigenschaftsgrenze auf einen früheren Zeitpunkt. Dabei müssen auch die rechten Grenzen aller Vorgängereigenschaften in die gleiche Richtung verschoben werden. Das hat dann wiederum zur Folge, dass andere Nachfolger dieser Vorgängereigenschaften mit verschoben werden müssen. Die Menge der Vorgänger und Nachfolger mit zu verschiebenden Grenzen erweitert sich so lange bis keine weiteren Eigenschaften betroffen sind. In vielen Fällen erhält man dabei die Menge aller Eigenschaften. Die Realisierbarkeit dieses Gedankengangs wurde in dieser Arbeit nicht näher untersucht.

Kombinatorische Schleifen

Der Wert einer Zusicherung hängt nur von Signalwerten zu früheren Zeitpunkten sowie dem aktuellen Zeitpunkt ab. Da der **next**-Operator nicht erlaubt ist, können spätere Zeitpunkte nicht auftreten. Die Abhängigkeit von früheren Zeitpunkten stellt mit dem in dieser Arbeit vorgestellten Syntheseverfahren kein Problem dar. Allerdings kann bei Abhängigkeit von Signalwerten des gegenwärtigen Zeitpunktes eine Situation entstehen, die genauer untersucht werden muss.

In Abb. 5.1 sind zwei Eigenschaften p und q dargestellt, die zu zwei willkürlichen Zeitpunkten Zusicherungen auf die Ausgangssignale $o1$ und $o2$ beinhalten. Eine mögliche Hardwareimplementierung zeigt Abb. 5.2, wobei die Realisierung mit Hilfe einer Multiplexerkaskade durchgeführt wurde. Besonders hervorgehoben ist dabei eine kombinatorische Schleife, welche auftreten kann, wenn beide aufgeführten Schaltsignale gleichzeitig aktiviert sind. Solche Schleifen sind in der Regel in synchroner Hardware nicht erwünscht und werden vom Logiksynthesetool aufgespürt.

In der Praxis wird es nicht vorkommen, dass beide Schaltsignale gleichzeitig aktiv sind, da sonst der Wert der Ausgangssignale nicht mehr eindeutig bestimmt ist. Es wird bei diesen Eigenschaften entweder der Determinie-

```
property p is
  ...
  prove:
    at t+4 : o2 = i+5;
    at t+4 : o1 = o2;
  ...
end property;

property q is
  ...
  prove:
    at t+2 : o2 = o1;
  ...
end property;
```

Abbildung 5.1: ITL-Quelltext von zyklischen Zusicherungen verschiedener Eigenschaften.

rungstest fehlschlagen oder die formale Prüfung der Eigenschaften auf dem generierten Design ist nicht erfolgreich.

Das Problem ist in *vhisyn* nur teilweise gelöst. Es existiert eine Routine, die bei der Implementierung jeder einzelnen Zusicherung prüft, ob eine solche Schleife entstanden ist. Wenn ja, wird der Nutzer entsprechend gewarnt.

Der Nutzer kann das Problem komplett umgehen, indem im Ausdruck einer Zusicherung keine Signale zum aktuellen Zeitpunkt verwendet werden, die an anderer Stelle im Eigenschaftssatz Ziel einer Zuweisung sind.

Konfliktzusicherungen und Eindeutigkeit der Annahmen

Zusicherungen auf das gleiche Signal können miteinander konkurrieren, wenn die entsprechenden Eigenschaftspositionen gleichzeitig vom Kontrollautomaten betreten werden. Es sind dabei zwei Fälle zu unterscheiden. Zum einen können die betroffenen Zusicherungen in jeder Situation dem Ausgangssignal die gleichen Werte zuweisen. Dann genügt es eine der Zusicherungen willkürlich auszuwählen. Wenn der für die Zuweisungen zuständige Mehrfachmultiplexer (alternativ auch eine Multiplexerkaskade) zwei oder mehr aktivierte Schaltsignale erhält, wird, je nach Implementierung, eines der Zuweisungssignale auf den Ausgang durchgeschalten. In dieser Situation gibt *vhisyn* eine entsprechende Warnung aus.

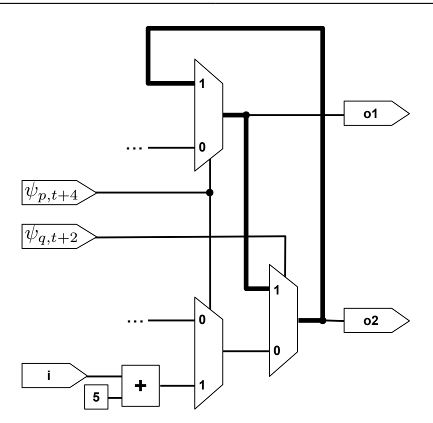


Abbildung 5.2: Mögliche Hardwareimplementierung der zu Abb. 5.1 gehörenden Zusicherungen.

Auf der anderen Seite können die parallelen Zusicherungen verschiedene Werte auf das selbe Signal zuweisen. Dies entspricht einem Konflikt, der entweder vom Determinierungstest oder einer Eigenschaftsprüfung des synthetisierten Entwurfs erkannt wird. Wenn einer der beiden Tests nicht durchgeführt wird, können parallele sich widersprechende Zuweisungen nur durch bereits genannte Warnung erkannt werden und die generierte Hardware ist möglicherweise fehlerhaft.

Dabei ist es wichtig zu verstehen, in welchen Situationen Zusicherungen aus verschiedenen Eigenschaften parallel aktiviert werden können. Diese Fragestellung entspricht der in Abschnitt 4.7.4 beantworteten Frage nach der Auflösung des Nichtdeterminismus von parallel verlaufenden Eigenschaften. Es lässt sich demnach feststellen, dass in parallel aktivierten Eigenschaften keine sich widersprechenden Zusicherungen vorkommen dürfen, bis der Zeitpunkt erreicht ist, an dem die Parallelität durch geeignete Annahmen aufgelöst wird.

5.1.5 Constraints

Ein wichtiger Bestandteil der formalen Verifikation ist die Bestimmung von Constraints, d.h. Beschränkungen des Umgebungsverhaltens der zu verifizierenden Komponente. Es werden dabei Ausdrücke formuliert, die nur die Eingänge der Schaltung benutzen dürfen und in jeder gültigen Situation zu

logisch wahr bzw. in jeder ungültigen Situation zu logisch falsch ausgewertet werden.

Wenn bspw. eine Komponente der Kommunikation mit einem Bus dient, ist ihr Verhalten bei ungültigen Busoperationen unter Umständen nicht spezifiziert. Das heisst, dass das resultierende willkürliche Verhalten nicht formal geprüft werden muss. Aus diesem Grund werden Constraints erstellt, die nur bei gültigen Busoperationen wahr und bei ungültigen Operationen falsch sind. Die formale Verifikation nutzt dann die Constraints um Fehler 1. Art (false positives) ausschließen zu können.

Bei der Synthese einer Eigenschaftbeschreibung werden Constraints nicht benötigt, da das Verhalten der Komponente bei ungültigem Umgebungsverhalten unwichtig ist. Es kann also auch fehlerhaft oder generell willkürlich erzeugt werden. Demnach ignoriert *vhisyn* in den Eigenschaften den Abschnitt **dependencies**, in dem die Constraints definiert werden, komplett.

5.2 Anforderungen an den Beschreibungstil

5.2.1 Kurze Operationseigenschaften

Der Beschreibungsstil mit Hilfe von Operationseigenschaften unterscheidet sich erheblich vom klassischen Entwurf auf RT-Ebene. Trotz dessen sind Analogien festzustellen. Anstelle eines zentralen Automaten tritt der Eigenschaftsgraph, während die Annahmen den Übergangsbedingungen entsprechen. Die Zusicherungen repräsentieren das Ausgangsverhalten, welches vom Automaten gesteuert wird.

Der große Vorteil einer Beschreibung mit Eigenschaften besteht darin, dass einzelne Eigenschaften das Verhalten des Systems über mehrere Takte hinweg modellieren können. Dabei entfällt die Aufgabe des expliziten Entwurfs des Automaten für die entstehenden Zwischenzustände. Es können alle Übergangsbedingungen und Zuweisungen, die zu einer Operation gehören, in einer Eigenschaft gekapselt werden, ohne dass sie über verschiedene Zustände des Automaten hinweg "verstreut" sind.

Aus diesem Vorteil ergibt sich jedoch eine weitere Einschränkung. Der Entwurf eines Systems, welches aus kleinen Eigenschaften besteht, lohnt sich nur sehr selten, wenn die Länge der Eigenschaften nur einen Takt beträgt, d.h. mit der klassischen Entwurfsmethodik wären keine Zwischenzustände nötig und einer der Vorteile des operationsbasierten Entwurfs geht verloren.

Ein erfahrener Designer, der beide Entwurfsparadigmen kennt, wird bei der Untersuchung einer Spezifikation feststellen können, ob sich eine Beschreibung mit Operationseigenschaften lohnt, oder doch besser auf die Möglichkeiten der RT-Ebene zurückgegriffen werden sollte.

5.2.2 Konzeptionelle Zustände

Bei der formalen Verifikation von Eigenschaften sind die sogenannten konzeptionellen Zustände (important states) von besonderer Bedeutung. Sie verknüpfen eine Eigenschaft, die sich ansonsten nur mit Ein- und Ausgangsverhalten beschäftigen muss, mit dem einer zu verifizierenden Komponente zugrunde liegenden Automaten. Die konzeptionellen Zustände nutzen üblicherweise interne Signale des Moduls, wie bspw. den Zustand des Automaten. Damit wird die formale Verifikation mit Operationseigenschaften zu einem Grey-Box-Verfahren.

An einem Beispiel wird dies deutlich. Wenn eine Kommunikationskomponente beginnend mit einer Startsequenz Rahmen fester Länge verarbeitet, so wird man eine Eigenschaft schreiben, die die Startsequenz im Annahmenteil beinhaltet und im Zusicherungsteil die fertig verarbeiteten Daten an den Ausgang legt. Wird nun der formale Beweis dieser Eigenschaft angetreten, erzeugt die Beweisengine ein Gegenbeispiel, bei dem sich die Komponente mitten in der Verarbeitung eines anderen Rahmens befindet, während die Startsequenz beginnt. Die Komponente reagiert dementsprechend falsch und der Beweis der Eigenschaft schlägt fehl.

Aus diesem Grund wird in den Annahmenteil der Eigenschaft der konzeptionelle Zustand ”bereit für Startsequenz” aufgenommen, der sicherstellt, dass die Eigenschaft nur in diesem Zustand beginnen darf. Um mehrere Eigenschaften miteinander zu verknüpfen muss im Zusicherungsteil ebenso ein konzeptioneller Zustand eingefügt werden, der dann bspw. sicherstellt, dass die Komponente wieder "bereit für Startsequenz” ist.

Bei der Synthese sind diese konzeptionellen Zustände nicht notwendig, denn die Vollständigkeit des Satzes von Eigenschaften kann auch bewiesen werden, wenn sie nicht auftauchen.

Wenn die Eigenschaften nach der Synthese auf dem erzeugten Entwurf geprüft werden sollen, müssen die konzeptionellen Zustände jedoch vorhanden sein. Aus diesem Grund wurde ein Verfahren eingesetzt, bei dem jeweils an den Grenzen einer Eigenschaft Platzhalter der Form **begin_eigenschaft** und **end_eigenschaft** per Hand eingefügt werden. Der Syntheseprozess ignoriert diese Platzhalter und erzeugt als Nebenprodukt der Hardware deren Definition in Form von ITL-Makros. Damit ist eine Verifikation des erzeugten Designs möglich.

Es sollte allerdings beachtet werden, dass diese Verifikation nur Fehler von *vhisyn* finden wird, da sowohl Eigenschaften als auch generiertes Design aus der gleichen Quelle stammen. Das bei der typischen formalen Verifikation von Eigenschaften so wichtige ”Vier-Augen-Prinzip” (Eigenschaften und Design werden getrennt erstellt) kann nicht angewendet werden. In Abschnitt

6.1.1 wird das Verfahren und der Prozess der Makrogenerierung genauer untersucht.

6 Ergebnisse

Dieses Kapitel beschreibt den Einsatz von *vhisyn* als Entwurfswerkzeug und die dabei erarbeiteten Ergebnisse. Zuerst wird die Verifikation eines erzeugten Designs anhand der zugrunde liegenden Eigenschaften erläutert. Ein weiterer Punkt ist das Zusammenspiel zwischen der High-Level-Synthese und der anschliessenden Logiksynthese. Es wird die bei der Logiksynthese erforderliche Funktionalität beschrieben, die nötig ist, um ITL-Beschreibungen effizient in Hardware umzusetzen.

Schließlich werden zwei Fallstudien vorgestellt, die unterschiedliche Aspekte eines Entwurfs beleuchten. Zum einen konnte eine ITL-Beschreibung eines Partikelfilters erstellt werden, die einen anspruchsvollen Datenfluss beinhaltet und aus mehreren Komponenten besteht, die ihrerseits jeweils vollständige Eigenschaftssätze darstellen.

Die zweite Fallstudie repräsentiert eine Framerkomponente, deren Spezifikation einem industriellen Umfeld entnommen wurde. Sie hat die Aufgabe in einem seriellen Datenstrom nach einem Synchronisationwort zu suchen und anhand dessen die Grenzen eines Datenrahmens zu erkennen. Dabei muss der Datenstrom auf die korrekte Bitposition ausgerichtet (*alignment*) und mit Hilfe einer Descramblingmethode dekodiert werden. Die Framerkomponente hat sehr lange Eigenschaften mit vielen Tausend Zuständen und verdeutlicht damit besonders gut die Vorteile der Konstruktion des deterministischen Kontrollautomaten und die dadurch ermöglichten Optimierungen der Zusammenfassung von Zusicherungen.

Den Schluss des Kapitels bildet eine kurze Beschreibung einer Visualisierungsplattform für die generierte Hardware. Sie liest Beschreibungen im .db Zwischenformat ein und zeigt ein Blockschaltbild der enthaltenen Hardware.

6.1 Formale Eigenschaftsprüfung des erzeugten Entwurfs

Das durch *vhisyn* generierte VHDL-Modell sollte, wenn *vhisyn* fehlerfrei arbeitet, exakt dem Verhalten der Eigenschaften entsprechen. Die formale Prüfung, ob das erzeugte Design die Eigenschaften erfüllt, wäre demnach nicht nötig. In der Praxis kann man jedoch einen Fehler in *vhisyn* nicht restlos ausschliessen. Aus diesem Grund kann eine formale Eigenschaftsprüfung er-

forderlich sein. Diese entspricht einem Äquivalenzvergleich zwischen Eigenschaftsmenge und erzeugtem RT-Design.

Weiterhin wurde in Abschnitt 3.6.4 bereits der fehlende Beweis der Implementierbarkeit einer Eigenschaftsmenge beschrieben. Eine formale Eigenschaftsprüfung des erzeugten Entwurfs mit den Eigenschaften, kann diese Art von Fehlern erkennen.

Abgesehen von der Implementierbarkeit können wirklich nur Fehler von *vhisyn* bzw. der zugrunde liegenden Methodik gefunden werden, und keine Entwurfsfehler, da Eigenschaften und erzeugtes Design voneinander abgeleitet werden (*single source*).

6.1.1 Fehlende konzeptionelle Zustände

Das große Problem bei dieser Prüfung liegt in den fehlenden konzeptionellen Zuständen von Eigenschaften, die in erster Linie für den Entwurf und nicht für die Verifikation geschrieben wurden. In Abschnitt 5.2.2 wurde bereits erwähnt, dass die Vollständigkeit eines Eigenschaftssatzes auch ohne konzeptionelle Zustände bewiesen werden kann [Bor09, Seite 97]. Aus diesem Grund werden Eigenschaften, die für den Entwurf geschrieben werden, keine konzeptionellen Zustände beinhalten. Bei der formalen Eigenschaftsprüfung sind sie jedoch nötig, um die Eigenschaft mit dem Design zu verknüpfen.

Die Lösung des Problems ist die Einführung von Platzhaltern in den Eigenschaften, die bei der High-Level-Synthese von *vhisyn* ignoriert werden, und bei der Eigenschaftsprüfung durch den entsprechenden konzeptionellen Zustand ersetzt werden. Wie in Abb. 6.1 zu ersehen ist, wird je an der linken und rechten Grenze der Eigenschaft ein solcher Platzhalter eingefügt, dessen Name sich aus dem Prefix **begin_** bzw. **end_** und dem Name der Eigenschaft zusammensetzt. Beim Einlesen der ITL-Dateien erkennt *vhisyn* genau diese beiden Bezeichner und ignoriert temporale Anweisungen, in denen nur dieser Ausdruck vorkommt. Nachteilig wirkt sich hierbei aus, dass diese Bezeichner dadurch vergeben sind und zur Beschreibung eines Designs nicht mehr zur Verfügung stehen.

Es stellt sich die Frage, womit diese Platzhalter ersetzt werden müssen. In einem ersten Versuch wurden beide Platzhalter durch das in Abschnitt 4.8 definierte Schaltsignal ersetzt. Dabei kommen die folgenden Entsprechungen zum Einsatz

$$\begin{aligned} begin_prop &:= \psi_{prop,left(prop)} \\ end_prop &:= \psi_{prop,right(prop)}, \end{aligned}$$

d.h. die Synthese erzeugt im Design zusätzliche Signale, die den Namen des Platzhalter haben und denen das entsprechende Schaltsignal zugewiesen

```
property prop is
  for timepoints:
    t_start = t+2,
    t_end   = t_start+2;

  assume:
    at t_start : begin_prop;
    ...

  prove:
    at t_end : end_prop;
    ...

  left_hook:  t_start;
  right_hook: t_end;
end property;
```

Abbildung 6.1: ITL-Eigenschaft mit Platzhaltern für die konzeptionellen Zustände.

wird. Mit dieser Vorgehensweise, können die Eigenschaften mit dem erzeugten Entwurf erfolgreich geprüft werden.

Allerdings taucht ein neues Problem auf. Der Nachfolgertest der Vollständigkeitsprüfung wird fehlschlagen, da er die erzeugte Beschreibung nicht verwenden darf und demnach nicht feststellen kann, dass das Ende der Vorgängereigenschaft den Beginn der Nachfolgereigenschaft impliziert. Formal ausgedrückt fehlt ihm das Wissen über

$$\overset{P}{\underset{p}{\forall}} \; \overset{succ(p)}{\underset{q}{\forall}} : \psi_{p,right(p)} \rightarrow \psi_{q,left(q)}, \tag{6.1}$$

um den konzeptionellen Endzustand mit dem gleichzeitigen konzeptionellen Startzustand der nächsten Eigenschaft in Beziehung zu setzen. Diese Erkenntnis führt direkt zur Lösung des Problems, indem **end_prop** umdefiniert wird, sodass es als Konjunktion der Startzustände aller Nachfolgereigenschaften definiert wird

$$end_prop := \bigwedge_{q}^{succ(prop)} \psi_{q,left(q)}.$$

Damit ist sowohl die Eigenschaftsprüfung erfüllt als auch der Nachfolgertest, der das Wissen aus Gleichung 6.1 nicht mehr benötigt.

Die im weiteren Verlauf dieses Kapitels vorgestellten beiden Fallstudien konnten mit Hilfe dieses Verfahrens die Eigenschaften erfolgreich am generierten Modell beweisen und somit die korrekte Funktionsweise von *vhisyn* exemplarisch nachweisen.

6.1.2 Fehlende Constraints

Ein weiteres Problem ist die Tatsache, dass beim Entwurf mit Operationseigenschaften keine Contraints notwendig sind, siehe Abschnitt 5.1.5. Um eine formale Eigenschaftsprüfung durchzuführen, sind diese jedoch wieder erforderlich und müssen vom Entwerfer bereitgestellt werden.

6.2 Zusammenspiel mit der Logiksynthese

Um aus Operationseigenschaften erzeugte Entwürfe auf Register-Transfer-Ebene benutzen zu können, müssen diese in einem Logiksyntheseschritt in eine Netzliste auf Gatterebene überführt werden. Damit das Ergebnis dieses Schrittes effizient ist, sind verschiedene Optimierungen notwendig. Sie können entweder direkt von *vhisyn* durchgeführt werden oder an die nachgelagerte Logiksynthese übertragen werden. In dieser Arbeit wurde sich für letzteres Vorgehen entschieden, um die sehr ausgereiften Optimierungsansätze moderner Synthesetools nutzen zu können. In späteren Arbeiten kann untersucht werden, ob die Anwendung dieser Ansätze auf der höhreren Ebene von *vhisyn* zusätzliche Vorteile ergibt, die zu kleineren und schnelleren Netzlisten führen.

6.2.1 Retiming

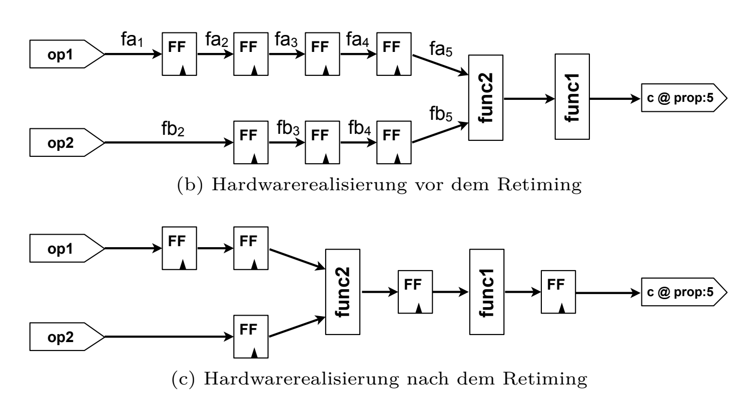
In diesem Abschnitt sollen insbesondere zwei Optimierungen angesprochen werden, die besonders vielversprechend bei Entwürfen sind, die aus Operationseigenschaften erzeugt wurden. Das erste Verfahren nennt sich Retiming und wurde erstmals von [LS91] beschrieben. Es ist heute Bestandteil der meisten Synthesetools.

Die Grundidee hinter Retiming ist ein geordnetes "Verschieben" von Registern, um die Taktfrequenz zu maximieren. Der originale und der durch Retiming veränderte Schaltkreis sind dabei funktional äquivalent. Das Verfahren wird in Abb. 6.2 illustriert. Die Eigenschaft in 6.2a wird direkt in die Hardware aus 6.2b abgebildet. Dabei kommt im gegebenen Beispiel die

```
property prop is
  ...
  freeze:
    fa = op1 @ t+1,
    fb = op2 @ t+2;

  ...
  prove:
    at t+5 : c = func1(func2((fa,fb));
end property;
```

(a) Fragment einer ITL-Eigenschaft



(b) Hardwarerealisierung vor dem Retiming

(c) Hardwarerealisierung nach dem Retiming

Abbildung 6.2: Illustration von Retiming anhand einer Eigenschaft mit Freezevariablen.

Registerkettenimplementierung von Freezevariablen zum Einsatz. Ohne Retiming entsteht dabei eine Kette von Registern, die die Eingänge der Schaltung um die gewünschte Anzahl von Takten verzögert. Danach schließen sich die beiden Datenflussoperationen **func1** und **func2** an, die rein kombinatorische Blöcke mit hoher Verzögerung darstellen. Es können nun unter Beibehaltung der ursprünglichen Funktionalität die Register der Kette durch die kombinatorischen Blöcke hindurch verschoben werden, wie bspw. in 6.2c zu erkennen ist. Dadurch wird der bisher sehr lange kombinatorische Pfad in kleinere Abschnitte unterteilt, sodass die maximale Taktfrequenz der Hardware steigt.

Da auf der einen Seite bei der Beschreibung von Eigenschaften Freezevariablen häufig genutzt werden, oder auch der **prev**-Operator mit hohen Verzögerungen zum Einsatz kommt, und auf der anderen Seite durch Makros sehr große rein kombinatorische Funktionsblöcke beschrieben werden, ist Retiming sehr wichtig, um effiziente RT-Hardware aus Operationseigenschaften erzeugen zu können. Alternativ kann man auch die Zuweisung in der Form

```
at t+5 : c = prev(func1(prev(func2(fa,fb))));
```

schreiben, um das selbe Ergebnis zu erzielen wie Retiming. Dies legt jedoch Implementierungsdetails auf einer niedrigen Ebene fest, was in einer High-Level-Beschreibung nicht erwünscht ist. Es ist zu empfehlen derartige Entscheidungen dem Logiksynthesetool zu überlassen, da es viel besser beurteilen kann, an welchen Stellen sich eine Optimierung lohnt (meist vor allem entlang des für die maximale Taktfrequenz kritischen Pfades).

6.2.2 Resource Sharing

Eine weitere extrem wichtige Optimierung stellt das sogenannte Resource Sharing dar. Dieses fasst gleichartige Schaltungsteile zusammen, wobei die Funktionalität erhalten bleibt. In ITL werden Makros oft mit den selben Argumenten in unterschiedlichen Situationen verwendet. Dazu zählen bspw. gleiche oder zumindest ähnliche Annahmen oder Zuweisungen in mehreren Eigenschaften. In diesen Fällen schreibt *vhisyn* die entstehende Hardware direkt in das Zwischenformat. Eventuelle Optimierungsmöglichkeiten werden nicht beachtet.

Die Methoden des Resource Sharing sind in weit entwickelter Form in allen modernen Synthesewerkzeugen integriert. Es muss lediglich darauf geachtet werden, dass sie bei der Synthese einer aus ITL generierten Beschreibung auch aktiviert werden.

6.3 Partikelfilter

6.3.1 Allgemeines

Partikelfilter sind Schätzverfahren aus der Klasse der Bayesfilter [TBF05]. Sie können im Gegensatz zu Kalmanfiltern beliebige Wahrscheinlichkeitsverteilungen repräsentieren. An der Professur Schaltkreis- und Systementwurf wurde im Projekt Generalisierte Plattform zur Sensordatenverarbeitung (GPSV) der BMBF-Initiative InnoProfile an der Positionsschätzung mit Hilfe eines Partikelfilters gearbeitet. Das dabei verwendete Testszenario umfasst ca. 20

fest installierte Funkknoten (Anker), die in den Laborräumen der Professur und im Aussengelände angebracht sind. Sie können ungefähr 20 Entfernungsmessungen pro Sekunde in Bezug zu einem mobilen Funkknoten durchführen. Damit ergibt sich eine geforderte Updatezeit des Filters von ungefähr 50 ms. Diese gemessenen Entfernungsangaben können mit Hilfe bekannter Ankerpositionen dazu dienen, die Position des mobilen Knotens zu schätzen. Das dabei benutzte Verfahren basiert auf einem Partikelfilter.

Der Filter enthält eine bestimmte Anzahl von Partikeln, die jeweils eine mögliche Position des Mobilknotens darstellen. Für jede durchgeführte Distanzmessung werden zwei Schritte durchgeführt. Im ersten Schritt (*update*) werden alle Partikel zufällig verschoben und einer Gewichtsberechnung zugeführt. Der Gewichtswert charakterisiert die Wahrscheinlichkeit, dass sich unter Annahme der aktuellen Messung der Partikel an derjenigen Position befindet, die sein Zustand ausdrückt. Die Partikelpositionen werden im zweiten Schritt (*resampling*) entsprechend ihrer Gewichtswerte vervielfältigt (hohes Gewicht) oder verworfen (niedriges Gewicht). Nach einer genügend großen Anzahl Messungen konzentrieren sich die Partikel in denjenigen Regionen, in denen die tatsächliche Position des Mobilknotens am wahrscheinlichsten ist.

Um ein gutes Schätzergebnis zu erreichen, werden ca. 8000 Partikel benötigt. Da eine Softwareverarbeitung von 8000 Partikeln auf dem Prozessor des mobilen Knotens ungefähr 660 ms dauert, können weniger als zwei komplette Filterdurchläufe pro Sekunde stattfinden. Die erreichbare Updatezeit ist damit deutlich niedriger als die geforderten 50 ms. Weiterhin ist in diesem Fall der Prozessor vollständig mit dem Abarbeiten des Schätzverfahrens beschäftigt und hat nicht genug Ressourcen für die Erfüllung anderer Aufgaben zur Verfügung. Aus diesem Grund wurde eine FPGA-Implementierung des Partikelfilters angestrebt, die als Koprozessor arbeitet, die Messungen entgegennimmt und die geschätzte Position des Mobilknotens zurückliefert. Für Details zur Implementierung des Filters wird an dieser Stelle auf [Fro+10] verwiesen.

6.3.2 Implementierung

Das Design wurde in ITL beschrieben und mit *vhisyn* in eine VHDL-Beschreibung auf RT-Ebene übersetzt. In Abb. 6.3 ist die gesamte Struktur dargestellt. Dabei wurden die Komponenten *resampler*, *updater*, *starter*, *random* und *timer* in ITL beschrieben, während die beiden FIFOs mit Xilinx Coregen erzeugt und per **external** Anweisung eingebunden wurden. Eine Besonderheit stellt die verschachtelte Instanziierung von Komponenten entsprechend Abschnitt 4.3.4 dar. So wird bspw. das Modul *random* sowohl in *starter* als auch in *updater* genutzt.

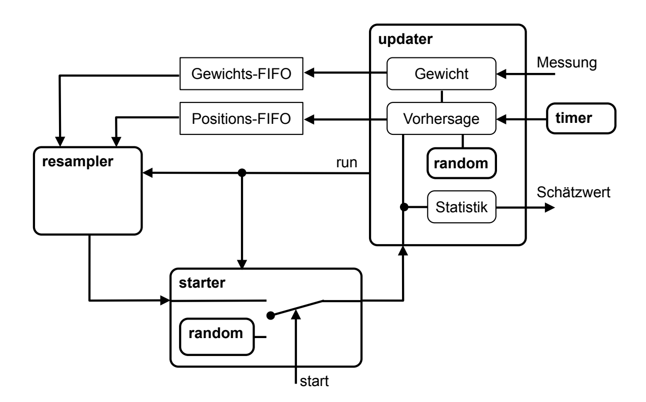


Abbildung 6.3: Leicht vereinfachte Struktur des Partikelfilters.

Die Größe der entstehenden VHDL-Datei beträgt 2,8 Megabyte und ist damit verhältnismäßig groß. Dies kommt dadurch zustande, dass jeder Operator, jede Instanz eines Funktionsaufrufs usw. als eigenständige parallele VHDL-Anweisung modelliert ist. Insbesondere rekursive Funktionen führen mit ihrer hohen Anzahl von Funktionsaufrufen zu ebenso vielen Hardwareblöcken, die den VHDL-Quellkode aufblähen.

Als Zielplattform dient ein Virtex-II Pro FPGA (xc2vp30) von Xilinx. Der generierten Beschreibung des Partikelfilters wurde eine Registerschnittstelle hinzugefügt, die über den PLB-Bus mit dem FPGA-internen PowerPC kommuniziert und u.a. die empfangenen Messungen in einem FIFO bereitstellt. In Abb. 6.4 ist das Gesamtsystem bestehend aus Partikelfilter, PowerPC und Host-PC dargestellt. Der Host-PC dient der grafischen Präsentation des Filterergebnisses und der Bereitstellung der vom Mobilknoten durchgeführten Entfernungsmessungen. Die Komponente *particle* enthält den Entwurf entsprechend Abb. 6.3.

In der prototypischen Plattform übernehmen PowerPC und Host-PC die Aufgabe die Messungen des Netzwerks aufzunehmen, an den Partikelfilter weiterzuleiten und anschliessend die geschätzte Position für den Nutzer aufzubereiten. Im konstruierten Anwendungsszenario sollen diese Aufgaben vom Prozessor des Mobilknotens übernommen werden.

Die gesamte VHDL-Beschreibung der Komponente *particle_shell* wurde mit Hilfe des Logiksynthesetools Synplify von Synopsys in eine Gatternetzliste übertragen, die im Anschluss mit Xilinx Plattform Studio 10 platziert und trassiert wurde. In Tab. 6.1 sind die enthaltenen Komponenten mit der

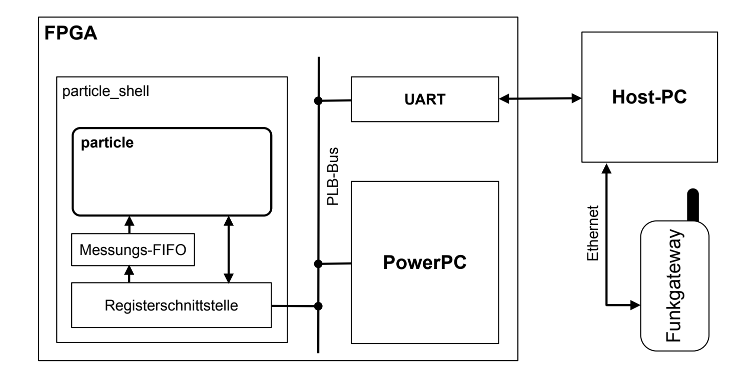


Abbildung 6.4: Integration des Partikelfilters in das Gesamtsystem.

Komponente	Eigenschaften	Kodezeilen	Zustände
resampler	5	248	8
updater	4	538	63
starter	6	220	8
random	2	90	3
timer	3	95	5
particle (Toplevel)	0	105	0

Tabelle 6.1: Überblick über die mit ITL beschriebenen Komponenten des Partikelfilters.

Anzahl der Eigenschaften und der durch die Automatenkonstruktion entstandenen Zustände aufgeführt.

Der synthetisierte Partikelfilter arbeitet mit einer maximalen Taktfrequenz von 8,6 MHz und benutzt ca. 47% der kombinatorischen Ressourcen des angegebenen FPGA. Er benötigt drei Takte um einen Partikel zu verarbeiten, woraus sich eine Updaterate von 2,9 ms ergibt. Dies unterschreitet die geforderte Updaterate von 50 ms um ein Vielfaches.

Betrachtungen zum Retiming

Die ITL-Implementierung insbesondere des *updater* setzt große Operatoren als rein kombinatorische Blöcke um. Aus diesem Grund ergibt sich ein sehr langer kritischer Pfad, der die maximale Taktfrequenz beschränkt. Um die Frequenz zu steigern, wurde das Logiksynthesetool angewiesen Retiming zu

benutzen, um verfügbare Register in die kombinatorischen Blöcke hineinzuschieben. Dafür darf die Anzahl Takte zwischen dem Einlesen eines Partikels und der Ausgabe nicht zu kurz sein, damit überhaupt Register vorhanden sind, die verschoben werden können. Dazu wurde die Verzögerung des Updaters als generischer Parameter **DELAY** definiert, der alle Zuweisungen an Ausgangssignale zum Zeitpunkt t_end + DELAY stattfinden lässt. Bei einem Wert von drei Takten wird dabei die höchste Taktfrequenz von 8.6 MHz erreicht.

Ohne Retiming wird eine Frequenz von lediglich 4,4 MHz erreicht. Die Anzahl der Register beträgt 4698, während sie mit Retiming um 43% darüber bei 6764 liegt. Der rein kombinatorische Logikaufwand steigt durch das Retiming um nur 5%.

Dabei tritt die Frage auf, warum höhere Werte kein noch besseres Retiming ermöglichen, und damit auch höhere Frequenzen. Dies ist darin begründet, dass mit höherer Verzögerung die Eigenschaften mehr überlappen, d.h. der Überlappungsfaktor entsprechend Abschnitt 4.6.5 wird sehr groß. Dies führt wiederum zu einem erheblich aufwendigeren Kontrollautomaten, der die Logiksynthese und damit auch das Retiming behindert.

6.3.3 Verifikation

Jede einzelne Komponente des Partikelfilters wurde mit dem formalen Verifikationswerkzeug 360MV geprüft. Es konnte durch die sukzessive Anwendung der Vollständigkeitsprüfung während der Entwicklung der einzelnen Komponenten sichergestellt werden, dass die Eigenschaften eine vollständige und eindeutige Beschreibung des gewünschten Verhaltens sind.

Bei der Prüfung des Moduls *updater* konnten einige Tests aufgrund von Komplexitätsproblemen nicht durchgeführt werden. Dies ist durch die komplexen Datenpfadoperationen in der Gewichtsberechnung bedingt [Fro+10].

Eine erfolgreiche Vollständigkeitsprüfung bedeutet jedoch noch nicht, dass auch das Gesamtsystem frei von Fehlern ist. So können im Zusammenspiel der einzelnen Module Fehler auftreten. Ein Fehler soll an dieser Stelle exemplarisch herausgegriffen werden.

Im konkreten Fall war ein sehr hartnäckiger Fehler im Resampler vorhanden, indem dieser am Ende eines kompletten Filterupdates aus den beiden FIFOs einige Partikel nicht ausgelesen hat, woraufhin diese im FIFO blieben. In der nächsten Runde wurden diese dann als erste Partikel ausgelesen, woraufhin die Gesamtsumme der Gewichte nicht mehr übereinstimmte.

Um diesen Fehler zu vermeiden, muss im System garantiert werden, dass beim Abschluss eines Filterupdates immer die korrekte Anzahl Partikel in

den FIFOs enthalten ist. Eine Eigenschaft der Form

$$run \rightarrow fifo.count = N$$

mit N als der Gesamtanzahl Partikel würde dies sicherstellen, lässt sich aber im allgemeinen Fall nur sehr schwer formal verifizieren. Verfahren wie in [Urd+10] beschrieben stellen eine Möglichkeit dar, die Vollständigkeit der Einzelmodule zu nutzen, um ein abstraktes Systemmodell zu erzeugen, auf welchem dann eine Eigenschaftsprüfung möglich ist. Eine alternative Fehlervermeidungsvariante stellt die kompositionale vollständige Verifikation im Sinne von [Bor09] dar. Mit ihr könnten Fehler dieser Art schon während des Entwurfs abfangen werden.

Beide Verfahren erfordern jedoch eine Beschreibung aller beteiligten Komponenten mit Hilfe vollständiger Eigenschaften. Dazu müssen auch für Module, die sehr verlässlich sind, wie bspw. die mit Coregen erzeugten FIFOs, vollständige Eigenschaftssätze vorliegen, was die Anwendung dieser Verfahren einschränkt.

Der Partikelfilter wurde nach erfolgtem Vollständigkeitsbeweis der Einzelkomponenten mit Hilfe klassischer simulationsbasierter Verfahren verifiziert. Dabei kam eine VHDL-Testbench zum Einsatz, mit deren Hilfe alle bisher aufgetretenen Systemfehler gefunden werden konnten. Fehler die nicht das Zusammenspiel mehrerer Module betrafen, sind in dieser Phase nicht aufgetreten, was die Nützlichkeit der Vollständigkeitsprüfung betont.

6.3.4 Vergleich mit alternativen Entwurfsmethoden

Um das Design dieser Komponente mit anderen Entwurfsmethoden zu vergleichen, wurde in [Sch10] die Spezifikation direkt als VHDL-Modell auf RT-Ebene implementiert und in einem weiteren Ansatz aus einer C-Beschreibung mit dem High-Level-Synthesewerkzeug CoDeveloper (vgl. [Imp10]) generiert. In diesem Abschnitt sollen die dabei gewonnenen Erkenntnisse kurz dargestellt werden. In [Lan+11] können weitere Details zum Vergleich nachgelesen werden. In Tab. 6.2 sind die Ergebnisse kurz zusammengefasst.

Die VHDL-Implementierung ist die dabei kleinste und auch flächensparendste Variante. Der Hauptgrund für die geringe Größe und vor allem hohe maximale Frequenz der synthetisierten VHDL-Beschreibung liegt in der Ausnutzung der mit den Xilinx-Werkzeugen gelieferten Coregen-Komponenten. So wurde ein für die Gewichtsberechnung notwendiger Dividierer mit einer sehr hohen Bitbreite von 48 Bit mit Coregen als effiziente Pipeline generiert, während die ITL-Variante lediglich den parallelen VHDL-Divisionsoperator verwendet. CoDeveloper kann im Gegensatz dazu Dividierer mit mehr als 32

	VHDL	***vhisyn***	**CoDeveloper**
Kodezeilen	2138 (vhdl)	1296 (vhi)	447 (C)
Geschätzter Zeitaufwand	1-2 Wochen	3 Tage	2 Tage
Slices	3855 (28%)	6011 (43%)	4603 (33%)
FlipFlops	5924 (21%)	5120 (18%)	5286 (19%)
4 Eingangs-LUT	3552 (12%)	8930 (32%)	6387 (23%)
BRAM	70 (51%)	69 (50%)	82 (60%)
MULT18x18	18 (13%)	23 (16%)	29 (21%)
Max. Frequenz (in MHz)	182	25	113
Takte pro Partikel	2	3	66
Updatezeit (in ms)	$0,09$	$0,98$	$4,8$

Tabelle 6.2: Vergleich der Syntheseergebnisse zwischen drei verschiedenen Entwurfsmethoden.

Bit nicht effizient schedulen, sodass der Durchsatz auf $\frac{1}{66}$ Partikel pro Takt abnimmt.

Ressourcen wie Multiplizierer, RAM-Speicher und FlipFlops lassen keine signifikanten Unterschiede zwischen den drei Implementierungen erkennen. Die Performance übertrifft in jedem Fall deutlich die eingangs angegebene Zeit von 50 ms pro Partikelupdate. Um einen schnellen Prototypen der Hardware zu erhalten, dessen Hardwareressourcen sich in der gleichen Größenordnung befinden, wie die einer manuellen Implementierung, ist der eigenschaftsbasierte Ansatz demnach geeignet.

6.3.5 Demonstrator

Die mit *vhisyn* erzeugte Hardwareimplementierung wurde mit Hilfe eines Funknetzwerks mit sechs Ankern erprobt. In diesem Versuch lief eine Person entlang einer festgelegten Route und trug dabei den mobilen Knoten mit sich. Die Entfernungsmessungen wurden über das Funknetzwerk an den Host-PC und von da weiter an das FPGA-Board geleitet, das dann die Position des Mobilknotens geschätzt hat. In Abb. 6.5 sind alle sechs Ankerpositionen auf einer Karte mit drei Laborräumen markiert. Weiterhin ist die geschätzte Route gegenüber der tatsächlichen dargestellt. Es kann festgestellt werden, dass der Filter korrekt funktioniert und eine mit *vhisyn* erzeugte Komponente erfolgreich auf einem FPGA ausgeführt werden kann. Der Host-PC, an den das FPGA-Board angeschlossen wurde, ist ebenfalls gekennzeichnet.

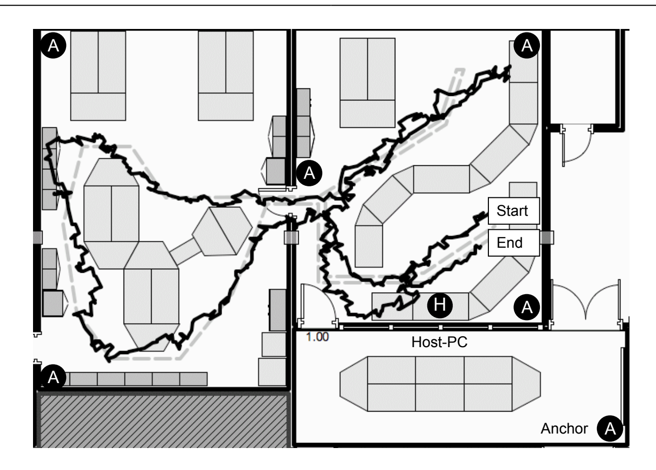


Abbildung 6.5: Testlauf des Partikelfilters in einem realen Anwendungsszenario mit sechs Ankern aus [Fro+10].

6.4 Framer

Als zweites umfassendes Beispiel für den Entwurf mit ITL dient eine Framer-Komponente. Dieses Beispiel war ursprünglich ein industrieller Entwurf, der bei Alcatel-Lucent verwendet wurde. In Kooperation mit diesem Unternehmen konnte ausgehend von der originalen Spezifikation eine eigenschaftsbasierte formale Verifikation durchgeführt werden [Sah+07]. Ausgehend von der dabei entstandenen vollständigen Eigenschaftsmenge, wurde ein neuer Satz Eigenschaften abgeleitet und mit *vhisyn* eine VHDL-Implementierung erzeugt. Dieser neue Eigenschaftssatz ist notwendig, da die originalen Eigenschaften speziell auf das zu verifizierende Design angepasst sind. So erfordert bspw. ein fehlendes Reset an mehreren internen Signalen zusätzliche Eigenschaften, die lediglich das "Einschwingverhalten" der Komponente beschreiben. Im neuen Eigenschaftssatz wurde sich exakt an der in der Spezifikation definierten Funktionalität orientiert.

Das in Abschnitt 4.6.1 eingeführte Beispiel ist eine sehr stark vereinfachte Variante dieser Komponente.

6.4.1 Funktionsweise

Der Framer hat mehrere Teilaufgaben zu erfüllen. In Abb. 6.6 ist das zugehörige Strukturbild dargestellt. Die enthaltenen Teilaufgaben werden im

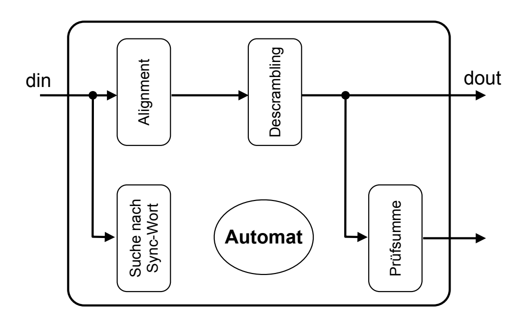


Abbildung 6.6: Aufbau und wichtige Bestandteile des Framers.

folgenden kurz erläutert. Zum einen sucht der Framer im eingehenden Datenstrom nach einem Synchronisationswort. Wenn dieses gefunden wird, merkt sich der Framer die genaue Bitposition. Nach Ablauf eines Datenrahmens von 19440 Worten wird an der selben Bitposition das Synchronisationswort erneut erwartet. Wenn es in einer bestimmten Anzahl von hintereinanderliegenden Rahmen an der gleichen Bitposition auftritt, wechselt der Framer von *idle* in den *sync* Modus. Nach einer Anzahl Rahmen, bei denen das Synchronisationswort nicht oder nicht an der richtigen Position gefunden wird, verlässt der Framer den *sync* Modus und begibt sich wieder auf die Suche nach einem Synchronisationswort. Dieses Verhalten konnte mit Hilfe von 13 Eigenschaften abgebildet werden. In Abb. 6.7 ist der Übersichtlichkeit halber der entsprechende Zustandsgraph abgebildet, dessen Kantengraph der Eigenschaftsgraph ist. Die Zustände dieses Graphen entsprechen den konzeptionellen Zuständen (siehe Abschnitt 3.6.7) und sind nicht mit den normalen Zuständen des bei der Synthese entstehenden Automaten vergleichbar. Die Übergänge des Graphen entsprechen den Eigenschaften. Alle, ausser denen, die im Zustand *lof* enden, sind ebenso lang wie ein Datenrahmen. Damit ergibt sich bei der Synthese mit *vhisyn* eine Anzahl von 175035 Automatenzuständen, was eine Hardwareabbildung in Multiple-Hot-Kodierung unpraktikabel macht. Unter Anwendung der in Abschnitt 4.9.3 beschriebenen Verfahren kann ein deterministischer Automat konstruiert und in Hardware abgebildet werden, der nur 146 explizite Zustandsübergänge besitzt.

Eine weitere Aufgabe des Systems ist das Alignment des Datenstromes. Dabei wird die bei der Suche nach dem Synchronisationswort gefundene Bitposition verwendet und einem Barrel-Shifter zugeführt, der den Datenstrom um die gewünschte Anzahl Bits verschiebt. Die korrekt ausgerichteten Datenworte werden anschliessend in einem Descrambler verarbeitet, der die Bits

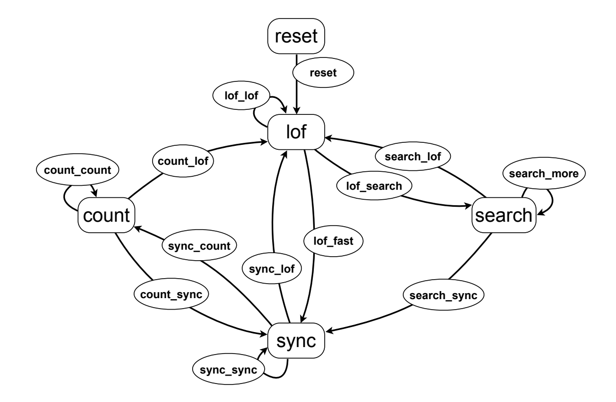


Abbildung 6.7: Zustandgraph des Framers.

mit einer vorher festgelegten Pseudozufallsfolge antivalent verknüpft (**xor**). In einem letzten Schritt wird byteweise eine Prüfsumme über einen Datenrahmen berechnet und kontrolliert. Wenn die Prüfsumme inkorrekt ist, wird dies nach aussen bekanntgegeben.

6.4.2 Vergleich mit Originaldesign

	Original-VHDL	***vhisyn***	**Faktor**
Anzahl Eigenschaften	29	13	45%
Kodezeilen	2009 (vhdl) + 2375 (vhi)	934 (vhi)	21%
Adaptive LUTs	703	1267	180%
FlipFlops	335	413	123%
Max. Frequenz (in MHz)	223	198	89%

Tabelle 6.3: Vergleich der Syntheseergebnisse zwischen originalem Framerdesign in VHDL und Eigenschaftsbeschreibung mit ITL.

In diesem Abschnitt soll der mit *vhisyn* durchgeführte Entwurf mit dem originalen Design in VHDL auf RT-Ebene verglichen werden. Die Ergebnisse sind in Tab. 6.3 abgebildet. Bei der Anzahl der Eigenschaften und Kodezeilen wird dabei beim Originalentwurf auch die Menge der für die Verifikation nötigen Eigenschaften in ITL berücksichtigt. Es ist beispielsweise auffällig,

dass bei der Verifikation mehr als doppelt so viele Eigenschaften nötig sind, als bei der direkten Nutzung als Entwurfssprache. Das ist zum einen darin begründet, dass die Verifikationseigenschaften aus Komplexitätsgründen nur eine geringe Länge aufweisen dürfen, und lange Eigenschaften mit mehreren kleinen nachgebildet werden müssen. Weiterhin bestehen im original VHDL-Design Initialisierungsprobleme, um die mit zusätzlichen Eigenschaften "herumgearbeitet" wurde.

Die Anzahl der Kodezeilen ist nicht exakt zu bestimmen, da in beiden Fällen zusätzliche Bibliotheken verwendet werden. So ist die Implementierung des Barrel-Shifter bspw. nicht in der Anzahl Kodezeilen enthalten.

Die Logiksynthese wurde mit Quartus II Version 9.1 für einen FPGA des Typs Stratix IV (EP4S100G5F45C2ES1) durchgeführt. Dabei verbraucht das erzeugte Design knapp die doppelte Anzahl Logikressourcen, während sich die Anzahl der FlipFlops jeweils in einem ähnlichen Rahmen bewegt. Die resultierende maximale Taktfrequenz von ca. 200 MHz ist in beiden Fällen höher als die geforderte Frequenz von 155MHz.

6.4.3 Laufzeit der Synthese

Um die Laufzeit von *vhisyn* insbesondere in Hinblick auf sehr lange Eigenschaften abschätzen zu können, wurde diese für verschiedene Rahmenlängen gemessen. In Abb. 6.8 ist sowohl die Gesamtlaufzeit der Synthese als auch die Zeit, die für die Konstruktion des Automaten aufgewendet werden muss, dargestellt. Die Messung der Gesamtzeit startet mit dem Aufruf von *vhisyn* mit allen benötigten ITL-Dateien und endet nachdem das VHDL-Modell geschrieben wurde. Die Zeit für die Automatenkonstruktion umfasst nur den Algorithmus entsprechend Abschnitt 4.7 und die zugehörige Abbildung des Automaten in Hardware.

Zusätzlich zu den tatsächlichen Messungen sind in der Abbildung zwei passende Trendlinien dargestellt, die mit Hilfe einer linearen Regression erstellt wurden. Mit der in der Legende angegebenen linearen Gleichung konnte ein Regressionskoeffizient R^2 von $0,9954$ bzw. $0,9937$ erreicht werden, was eine hohe "Passgenauigkeit" der linearen Gleichung bedeutet. Die in Abschnitt 4.7.7 ermittelte lineare Komplexität des Konstruktionsalgorithmus in Abhängigkeit von der Länge der Eigenschaften wird damit praktisch nachgewiesen, d.h. in der Praxis auftretende Eigenschaftslängen von mehreren Tausend Takten stellen für den Algorithmus kein Problem dar.

Die Laufzeitmessungen wurden auf einem Linux-Rechner mit zwei Dual-Core AMD Opteron Prozessoren mit je 2,6 GHz und insgesamt 16 GB Hauptspeicher durchgeführt. Da *vhisyn* nicht nebenläufig arbeitet, kann nur einer der Prozessoren zur gleichen Zeit genutzt werden.

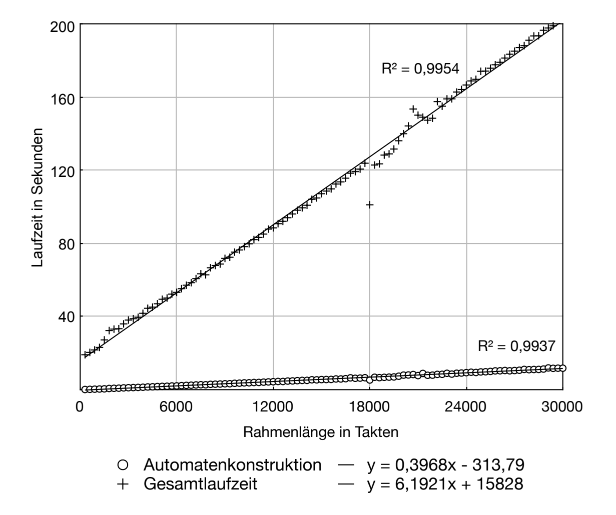


Abbildung 6.8: Laufzeit der Synthese des Framers mit unterschiedlichen Rahmenlängen.

6.4.4 Verbesserungen

In dieser Fallstudie ist es ein Nachteil von Beschreibungen mittels ITL-Eigenschaften, dass gleiches oder ähnliches Verhalten in allen Operationen, die dieses Verhalten auslösen, immer wieder beschrieben werden muss. Zur Verminderung dieses Problems werden Makros eingesetzt, die durch mehrmaligen Aufruf sowohl den Beschreibungs- als auch Wartungsaufwand des Kodes verringern. Eine weitere Möglichkeit sind temporale Makros, die es erlauben mehrere Signalzuweisungen zu gruppieren. In *vhisyn* sind temporale Makros allerdings nicht implementiert, sodass dieses Feature im Framer nicht verwendet werden kann.

Eine dritte Möglichkeit eine besser wartbare und kompaktere Beschreibung zu erstellen, wären abstrakte Eigenschaften, die mit Hilfe von Ableitung anderen Eigenschaften Funktionalität bereitstellen können. Sowohl Ableitungen, als auch temporale Makros können synthetisiert werden, ohne dass die grundlegenden Algorithmen in *vhisyn* davon beeinflusst werden.

6.5 Fazit

Insgesamt lässt sich feststellen, dass die Ergebnisse, die mit *vhisyn* erzielt werden, in Bezug auf Ressourcenverbrauch und maximal erreichbare Taktfrequenz in der selben Größenordnung liegen, wie manuell erstellte Entwürfe auf einer niedrigeren Ebene. Die verbleibenden Unterschiede können zum Teil durch Optimierungen vermindert werden, müssen jedoch teilweise hingenommen werden, da man auf der höheren Ebene nicht die gleichen Möglichkeiten speziell angepasster Verfahren hat wie auf niedrigeren Ebenen.

Die Vorteile des operationsbasierten Entwurfs liegen klar auf der Hand. Die Entwurfsebene befindet sich zwischen Register-Transfer- und algorithmischer Ebene und ist deshalb zum einen für Anwendungsgebiete, die taktgenaue Schnittstellen erfordern sehr gut geeignet. Zum anderen kommt die Beschreibung der Funktionalität mit Hilfe von Operationen den in der Spezifikation verwendeten Zeitdiagrammen sehr nahe und senkt damit den Entwurfsaufwand, der durch Verfeinerungsschritte entsteht.

Den größten Vorteil bietet jedoch die Vollständigkeitsprüfung der Eigenschaftsbeschreibung. Sie kann bereits in einem frühen Entwurfsstadium in den Designprozess einbezogen werden und hilft dabei, Fehler frühzeitig zu vermeiden.

6.6 Visualisierung des Zwischenformats

Die Beschreibung eines Entwurfs mit Hilfe von Operationseigenschaften erfolgt auf einer Ebene, die sich zwischen Register-Transfer- und algorithmischer Ebene befindet. Da der Entwerfer die interne Funktionsweise von *vhisyn* nicht im Detail kennt, kann es sich als nützlich erweisen, wenn er Zugriff auf die generierten Strukturen auf RT-Ebene hat. Der erzeugte VHDL-Kode ist jedoch sehr umfangreich und unübersichtlich. Aus diesem Grund wurde im Rahmen dieser Arbeit eine Visualisierungsplattform für das Zwischenformat entsprechend Abschnitt 4.1 geschaffen. Sie liest eine *.db*-Datei mit einem erzeugten Modell ein und stellt die Strukturen graphisch mit der Visualisierungbibliothek Nlview™ von Concept Engineering dar [Con11].

Das bereits bekannte Beispiel des Makros **seq_prev** wird in Abb. 6.9 durch Nlview dargestellt. Über jedem Block steht dabei der Name der Instanz im Format

_inst_l<*Zeile*>_c<*Spalte*>_<*Id*>,

wobei Zeile und Spalte die Stelle des Tokens im Quelltext kennzeichnen, das diesem Block entspricht.

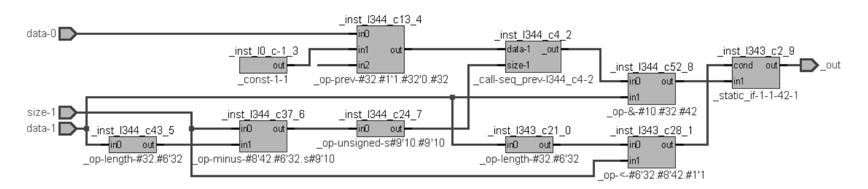


Abbildung 6.9: Visualisierung einer Instanz des Makros **seq_prev** aus Abb. 4.16 und Abb. 4.17a mit Hilfe der Nlview™Bibliothek.

Unterhalb des Blocks wird der Name des Moduls dargestellt. Das Präfix **_op** kennzeichnet grundlegende Hardwareoperatoren. Der Bezeichner beinhaltet in diesem Fall nicht nur den Namen des Operators, sondern auch die Breite und Signedness der Argumente und des Rückgabetyps. Falls einer dieser Werte zur Synthesezeit als Konstante feststeht, ist auch diese Information im Bezeichner kodiert. Weitere Blocktypen sind Konstanten (**_const-<*Breite*>-<*Wert*>**), Makroaufrufe (**_call-<*Modulname*>**) und statische Verzweigungen (Präfix **_static-if**).

In Abb. 6.10 ist ein weiteres Beispiel gegeben. Die Eigenschaft *random_reset* enthält vier Zusicherungen in zwei **during**-Anweisungen.

Durch Zusammenfassen entsprechend Abschnitt 4.5.6 wurden die Äquivalenzen

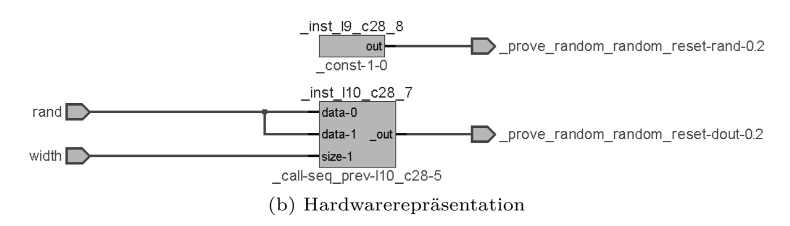
$$z_{random_reset,rand,0} \sim z_{random_reset,rand,1}$$
$$z_{random_reset,dout,0} \sim z_{random_reset,dout,1}$$

festgestellt und die jeweiligen Zusicherungen auf einen Ausgang gelegt. Demnach wird bspw. die Hardware für den Aufruf von *seq_prev* nur einmal instanziiert. Das rechtsoffene Intervall am Schluss des Ausgangsbezeichners gibt an, für welche Zeitpunkte des betreffenden Signals die Zusicherung gilt.

Durch die exakte Bezeichnung von Modulen und Instanzen im Zwischenformat ist es möglich, für jeden Block auf RT-Ebene das entsprechende ITL-Konstrukt zu finden. Die mit *vhisyn* durchgeführte Synthese ist somit vollständig transparent. Dies erlaubt die Realisierung von weiteren Werkzeugen, die das Arbeiten mit ITL erleichtern, wie zum Beispiel einem speziellen Simulator, der die Ergebnisse der Simulation des generierten Entwurfs ansprechend visualisiert und bessere Debuggingmöglichkeiten bietet.

```
property random_reset is
    for timepoints:
        t_end    = t+1;
    assume:
        reset_sequence;
    prove:
        during [t,t_end] : rand = 0;
        during [t,t_end] : dout = seq_prev(rand,WIDTH);
    right_hook: t_end;
end property;
```

(a) ITL-Eigenschaft *random_reset*



(b) Hardwarerepräsentation

Abbildung 6.10: Visualisierung der Hardwarerepräsentation einer Eigenschaft *random_reset* mit Hilfe der Nlview™Bibliothek.

7 Schluss

7.1 Zusammenfassung

In dieser Arbeit wurde eine durchgehende Synthesemethodik beschrieben, die eine vollständige Menge von Operationseigenschaften in ein Modell auf RT-Ebene umsetzt. Operationseigenschaften wurden bisher in der formalen Verifikation eingesetzt. Sie besitzen einen einfachen Aufbau und können entsprechend eines Eigenschaftsgraphen aneinandergereiht werden, um das Verhalten des Systems zu jedem Zeitpunkt eindeutig zu bestimmen. In der formalen Verifikation werden zum einen die Eigenschaften auf der Implementierung geprüft. Zum anderen wird mit Hilfe einer Vollständigkeitsprüfung nachgewiesen, dass die Eigenschaften keine Lücken im Systemverhalten aufweisen. Durch diese Prüfung stellt die Eigenschaftsmenge ein Referenzdesign dar, welches in seinem Ein-/Ausgangsverhalten exakt dem realisierten Entwurf entspricht.

Es ist demnach möglich aus diesem Referenzdesign eine Implementierung auf RT-Ebene automatisiert herzuleiten und Operationseigenschaften als Entwurfsmethodik zu etablieren. In dieser Arbeit wurden Algorithmen vorgestellt und untersucht, um aus dem Eigenschaftsgraph einer vollständigen Beschreibung einen nichtdeterministischen oder deterministischen Kontrollautomaten zu konstruieren. Weitere Untersuchungen zeigen die effiziente Abbildung der wesentlichen Konstrukte von ITL.

Die High-Level-Synthese aus einer Eigenschaftsmenge bietet zwei wesentliche Vorteile. Zum einen hilft der Beweis der Vollständigkeit dabei Fehler in der Entwurfsphase zu vermeiden, zum anderen ist der Entwurf eines Systems mit Hilfe von Operationen in verschiedenen Anwendungsbereichen sehr vorteilhaft. Insbesondere Komponenten, die Protokolle mit langen Sequenzen implementieren, können durch Operationseigenschaften sehr komfortabel dargestellt werden. Neben dem Operationsgedanken stellt der Entwurf der Datenflussfunktionalität mit ITL-Makros und ihrem funktionalen Beschreibungsstil im Gegensatz sowohl zu algorithmischen Beschreibungen aus imperativen Programmiersprachen wie C oder Matlab als auch zu den eingeschränkten Möglichkeiten auf RT-Ebene eine komfortable Alternative dar.

Das im Rahmen dieser Arbeit entstandene prototypische Entwurfswerkzeug *vhisyn* wurde anhand zweier komplexer Fallstudien untersucht und die

Ergebnisse hinsichtlich ihrer Ressourcenauslastung und maximalen Taktfrequenz verglichen. Auch die Laufzeit der Synthese für unterschiedliche Eigenschaftslängen sowie der Ressourcenverbrauch im Vergleich zu alternativen Beschreibungsmethoden wurden untersucht.

7.2 Ausblick

Weiterführende Arbeiten sollten versuchen die Anforderungen an eine synthesefähige ITL-Beschreibung zu lockern. Insbesondere Intervalle in Zeitvariablen sind mit der vorliegenden Methodik prinzipiell umzusetzen und bieten die Möglichkeit Eigenschaften variabler Länge zu verwenden. Dies ist insbesondere für die Beschreibung von Wartezyklen in Kommunikationsprotokollen sehr hilfreich. Damit vergrößert sich die Menge an Anwendungen, in denen Operationseigenschaften eine sehr effiziente Lösung darstellen. Weiterhin kann die Nutzerakzeptanz unter den Entwerfern erhöht werden, wenn einige der nicht unterstützten Operatoren, wie temporale Makros, **foreach** oder **let**, zur Verfügung stehen.

Intensive Untersuchungen sind notwendig, um nicht "vorhersagende" Verwendungen des **next**-Operators aufzulösen oder Zusicherungen zu unterstützen, die nicht in der Form **out** = *Ausdruck* vorliegen. In den entsprechenden Abschnitten wurden dazu bereits mögliche Lösungsansätze skizziert.

Weiteres Verbesserungspotential bietet die Optimierung bzw. Wiederverwendung funktional gleicher Ausdrücke oder Teilausdrücke. Durch eine Optimierung auf ITL-Ebene könnte das Logiksynthesewerkzeug entlastet werden und das generierte VHDL-Design übersichtlicher und kleiner werden. Zum Teil wurde dies für Zusicherungen innerhalb von **during**-Anweisungen im Zusicherungsteil bereits realisiert, um sehr lange Eigenschaften überhaupt unterstützen zu können. Eine anweisungs- bzw. eigenschaftsübergreifende Optimierung ist jedoch in der vorliegenden Version von *vhisyn* nicht implementiert.

Bei der Vorstellung der beiden Fallstudien wurden bereits einige weitere Verbesserungen vorgeschlagen, die den Entwurfsaufwand reduzieren und damit besser wartbare und weniger fehleranfällige Beschreibungen ermöglichen. So könnte Vererbung dabei helfen, gleiche Funktionalität in eine Basiseigenschaft auszulagern. Entwurfsaufwand und Wartbarkeit des Quelltextes könnten auf diese Weise erheblich verbessert werden.

Der Vollständigkeitsbeweis kann die Implementierbarkeit des Eigenschaftssatzes nicht nachweisen. Dies führt zu Problemen, wenn zwei Eigenschaften in ihrer nichtdeterministischen Phase, d.h. bevor die Annahmen eine der beiden Eigenschaften beenden, widersprüchliche Zusicherungen aufweisen. Das wird

weder vom Determinierungstest noch von einem der anderen Vollständigkeitstests entdeckt. Man spricht in diesem Fall davon, dass die Eigenschaftsmenge nicht implementierbar ist.

Die Synthese geht jedoch von einer Implementierbarkeit aus und warnt den Nutzer lediglich, dass zwei möglicherweise parallel ausgeführte Zusicherungen auftreten. In der Realisierung wird sich dann willkürlich für eine der beiden Zusicherungen entscheiden. Die Auswahl hängt von der Reihenfolge im Ausgangsmultiplexer ab. Die fehlende Übereinstimmung zwischen nicht implementierbarem Eigenschaftssatz und Implementierung tritt erst dann zutage, wenn die Eigenschaften auf dem generierten Design geprüft werden und mindestens eine der Eigenschaften nicht erfüllt ist. Der in dieser Arbeit vorgeschlagene Entwurfsfluss entdeckt also den Fehler. Es ist jedoch vorteilhaft, wenn die fehlende Implementierbarkeit möglichst frühzeitig während des Entwurfs detektiert wird und eine als vollständig und implementierbar nachgewiesene Eigenschaftsmenge die Erzeugung eines korrekten Designs garantiert.

Ein zusätzlicher formaler Implementierbarkeitstest kann dieses Problem lösen und ist Gegenstand zukünftiger Untersuchungen.

Literaturverzeichnis

[Aba+00] Yael Abarbanel, Ilan Beer, Leonid Gluhovsky, Sharon Keidar und Yaron Wolfsthal. „FoCs - Automatic Generation of Simulation Checkers from Formal Specifications“. In: *Computer Aided Verification*. Springer, 2000, S. 538–542.

[Acc11] *Accellera*. 2011. URL: http://www.accellera.org (besucht am 02. 05. 2011).

[AL08] Peter J. Ashenden und Jim Lewis. *The Designer's Guide to VHDL*. 3rd Ed. Morgan Kaufmann, 2008. ISBN: 978-0-12-088785-9.

[Arm+02] Roy Armoni, Limor Fix, Alon Flaisher, Rob Gerth, Boris Ginsburg, Tomer Kanza, Avner Landver, Sela Mador-Haim, Eli Singerman, Andreas Tiemeyer, Moshe Y. Vardi und Yael Zbar. „The ForSpec Temporal Logic: A New Temporal Property-Specification Language“. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2002, S. 296–311.

[Arm+06] Roy Armoni, Dmitry Korchemny, Andreas Tiemeyer, Moshe Y. Vardi und Yael Zbar. „Deterministic Dynamic Monitors for Linear-Time Assertions“. In: *Formal Approaches to Testing and Runtime Verification (FATES/RV)*. Springer, 2006, S. 163–177.

[Bac78] John Backus. „Can Programming be Liberated from the von Neumann Style?: A Functional Style and its Algebra of Programs“. In: *Communications of the ACM* 21.8 (Aug. 1978), S. 613–641. ISSN: 00010782.

[BH97] Bishop C. Brock und Warren A. Hunt Jr. „The DUAL-EVAL Hardware Description Language and Its Use in the Formal Specification and Verification of the FM9001 Microprocessor“. In: *Formal Methods in System Design* 11.1 (Juli 1997), S. 71–104. ISSN: 0925-9856.

[Bie+99] Armin Biere, A. Cimatti, Edmund M. Clarke und Yunshan Zhu. „Symbolic Model Checking without BDDs“. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 1999, S. 193–207.

[BL69] J. Richard Büchi und Lawrence H. Landweber. „Solving Sequential Conditions by Finite-State Strategies“. In: *Transactions of the American Mathematical Society* 138 (1969), S. 295–311.

[Blo+07] Roderick Bloem, Stefan Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli und Martin Weiglhofer. „Automatic Hardware Synthesis from Specifications: A Case Study“. In: *Design, Automation, and Test in Europe (DATE)*. 2007, S. 1188–1193.

[BMP81] Mordechai Ben-Ari, Zohar Manna und Amir Pnueli. „The Temporal Logic of Branching Time“. In: *Principles of Programming Languages (POPL)*. New York, NY, USA: ACM, Jan. 1981, S. 164–176. ISBN: 089791029X.

[Bor09] Jörg Bormann. „Vollständige funktionale Verifikation“. Dissertation. Universität Kaiserslautern, 2009.

[Bry86] Randal E. Bryant. „Graph-Based Algorithms for Boolean Function Manipulation“. In: *IEEE Transactions on Computers* 35.8 (1986), S. 677–691.

[Bur+90] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill und L. J. Hwang. „Symbolic model checking: 10ˆ20 states and beyond“. In: *Logic in Computer Science (LICS)*. IEEE, 1990, S. 428–439. ISBN: 0-8186-2073-0.

[BZ08] Marc Boulé und Zeljko Zilic. *Generating Hardware Assertion Checkers: for Hardware Verification, Emulation, Post-Fabrication Debugging and On-Line Monitoring*. Springer, 2008, S. 279–. ISBN: 1402085850.

[Cer+98] Eduard Cerny, Bachir Berkane, Pierre Girodias und Karim Khordoc. *Hierarchical Annotated Action Diagrams*. Kluwer, 1998, S. 232–. ISBN: 079238301X.

[CGP99] Edmund M. Clarke, Orna Grumberg und Doron Peled. *Model checking*. MIT Press, 1999, S. 314–. ISBN: 0262032708.

[Cha+03] Kai-Hui Chang, Wei-Ting Tu, Yi-Jong Yeh und Sy-Yen Kuo. „A Simulation-Based Temporal Assertion Checker for PSL“. In: *Midwest Symposium on Circuits and Systems (MWSCAS)*. IEEE, 2003, S. 1528–1531.

[Chu32] Alonzo Church. „A Set of Postulates for the Foundation of Logic“. In: *The Annals of Mathematics* 33.2 (1932), S. 346–366.

[Chu63] Alonzo Church. „Logic, Arithmetic, and Automata“. In: *International Congress of Mathematicians*. 1963, S. 23–35.

[CKV01] Hana Chockler, Orna Kupferman und Moshe Y. Vardi. „Coverage Metrics for Temporal Logic Model Checking“. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, Apr. 2001, S. 528–542. ISBN: 3-540-41865-2.

[Cla+01] Edmund M. Clarke, Armin Biere, Richard Raimi und Yunshan Zhu. „Bounded Model Checking Using Satisfiability Solving“. In: *Formal Methods in System Design* 19.1 (Juli 2001), S. 7–34.

[Cla01] Koen Claessen. „Embedded Languages for Describing and Verifying Hardware“. Dissertation. Chalmers University of Technology und Göteborg University, 2001.

[CLM98] Byron Cook, John Launchbury und John Matthews. „Specifying Superscalar Microprocessors in Hawk“. In: *Formal Techniques for Hardware and Hardware-like Systems (FTH)*. Marstrand, Schweden, 1998.

[Con11] *Concept Engineering*. 2011. URL: http://www.concept.de (besucht am 07. 01. 2011).

[CS00] Koen Claessen und Mary Sheeran. *A Tutorial on Lava: A Hardware Description and Verification System*. Techn. Ber. Chalmers University of Technology, 2000.

[Die10] Reinhard Diestel. *Graph Theory*. 4th Ed. Bd. 173. Heidelberg: Springer, 2010, S. 451–. ISBN: 9783642142789.

[DLL62] Martin Davis, George Logemann und Donald Loveland. „A Machine Program for Theorem-Proving“. In: *Communications of the ACM* 5.7 (Juli 1962), S. 394–397.

[DP60] Martin Davis und Hilary Putnam. „A Computing Procedure for Quantification Theory“. In: *Journal of the ACM (JACM)* 7.3 (1960), S. 201–215.

[EH82] E. Allen Emerson und Joseph Y. Halpern. „Decision procedures and expressiveness in the temporal logic of branching time“. In: *Symposium on Theory of Computing (STOC)*. New York, New York, USA: ACM, 1982, S. 169–180. ISBN: 0897910702.

[EH86] E. Allen Emerson und Joseph Y. Halpern. „“Sometimes” and “not never” revisited: on branching versus linear time temporal logic“. In: *Journal of the ACM (JACM)* 33.1 (Jan. 1986), S. 151–178. ISSN: 00045411.

[Eme08] E. Allen Emerson. „The Beginning of Model Checking: A Personal Perspective“. In: *25 Years of Model Checking*. Hrsg. von Orna Grumberg und Helmut Veith. Bd. 5000. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, Jan. 2008, S. 27–45. ISBN: 978-3-540-69849-4.

[Eme90] E. Allen Emerson. „Temporal and Modal Logic“. In: *Handbook of Theoretical Computer Science (Vol. B)*. Hrsg. von Jan van Leeuwen. MIT Press, 1990. Kap. 16, S. 995–1072. ISBN: 0262720140.

[Fic83] Faith E. Fich. „New Bounds for Parallel Prefix Circuits“. In: *Symposium on Theory of Computing (STOC)*. New York, NY, USA: ACM, 1983, S. 100–109. ISBN: 0897910990.

[FKL04] Harry D. Foster, Adam C. Krolnik und David J. Lacey. *Assertion-Based Design*. 2nd Ed. Kluwer, Mai 2004. ISBN: 1402080271.

[Fro+10] Daniel Froß, Jan Langer, André Froß, Marko Rößler und Ulrich Heinkel. „Hardware Implementation of a Particle Filter for Location Estimation“. In: *Indoor Positioning and Indoor Navigation (IPIN)*. Zurich, Switzerland: IEEE, 2010, S. 1–6.

[Ger99] Sabih H. Gerez. *Algorithms for VLSI design automation*. Chichester: John Wiley & Sons, 1999.

[GHS03] Mike Gordon, Joe Hurd und Konrad Slind. „Executing the formal semantics of the Accellera Property Specification Language by mechanised theorem proving“. In: *Correct Hardware Design and Verification Methods (CHARME)*. Springer, 2003, S. 200–215.

[GK83] Daniel D. Gajski und Robert H. Kuhn. „New VLSI Tools“. In: *Computer* 16.12 (Dez. 1983), S. 11–14. ISSN: 0018-9162.

[HA04] James C. Hoe und Arvind. „Operation-Centric Hardware Description and Synthesis“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 23.9 (Sep. 2004), S. 1277–1288. ISSN: 0278-0070.

[Hei+03] Ulrich Heinkel, Claus Mayer, Jörg Pleickart, Joachim Knäblein, Hans Sahm, Charles Webb, Werner Haas und Stefan Goßens. „Specification, Design and Verification of Systems-on-Chip in a Telecom Application“. In: *Entwurf Integrierter Schaltungen und Systeme (E.I.S.)* Erlangen: VDE, 2003, S. 45–50.

[Hei99] Ulrich Heinkel. „Formale Spezifikation und Validierung digitaler Schaltungsbeschreibungen mit Zeitdiagrammen“. Dissertation. Universität Erlangen-Nürnberg, 1999.

[HMU06] John E. Hopcroft, Rajeev Motwani und Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* 3rd. Pearson, 2006.

[Hoa03] Charles Antony Richard Hoare. „Assertions: A personal perspective“. In: *IEEE Annals of the History of Computing* 25.2 (Apr. 2003), S. 14–25. ISSN: 1058-6180.

[Hud+92] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain und John Peterson. „Report on the Programming Language Haskell: A Non-strict, Purely Functional Language“. In: *ACM SIGPLAN Notices - Haskell special issue* 27.5 (1992), S. 1–164.

[IEEE01] Institute of Electrical and Electronics Engineers. *IEEE Std 1364-2001 - IEEE Standard Verilog Hardware Description Language.* Techn. Ber. 2001.

[IEEE04] Institute of Electrical and Electronics Engineers. *IEEE Std 1076.6-2004 - IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis.* Techn. Ber. 2004.

[IEEE05a] Institute of Electrical and Electronics Engineers. *IEEE Std 1800-2005 - IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language.* Techn. Ber. 2005.

[IEEE05b] Institute of Electrical and Electronics Engineers. *IEEE Std 1850-2005 - IEEE Standard for PSL - Property Specification Language.* Techn. Ber. 2005.

[IEEE08] Institute of Electrical and Electronics Engineers. *IEEE Std 1647-2008 - IEEE Standard for the Functional Verification Language e.* Techn. Ber. 2008.

[IEEE09] Institute of Electrical and Electronics Engineers. *IEEE Std 1076-2008 - IEEE Standard VHDL Language Reference Manual.* Techn. Ber. 2009.

[Imp10] *Impulse Accelerated Technologies.* 2010. URL: http://www.impulseaccelerated.com (besucht am 01. 12. 2010).

[Kar72] Richard M. Karp. „Reducibility Among Combinatorial Problems“. In: *Complexity of Computer Computations.* Hrsg. von R E Miller und J W Thatcher. New York, NY: Plenum Press, 1972, S. 85–103. ISBN: 978-3-540-68274-5.

[Kup+10] Jan Kuper, Christiaan Baaij, Matthijs Kooijman und Marco Gerards. „Exercises in architecture specification using CλaSH“. In: *Forum on Specification & Design Languages (FDL).* Giéres, France, 2010, S. 178–183.

[Lan+11] Jan Langer, Daniel Froß, Enrico Billich, Marko Rößler und Ulrich Heinkel. „Multi-Level Synthesis on the Example of a Particle Filter“. In: *Southern Conference on Programmable Logic (SPL) - Designer Forum.* Cordoba, Argentina, 2011. ISBN: 978-84-614-7682-4.

[Leh74] Philippe G. H. Lehot. „An Optimal Algorithm to Detect a Line Graph and Output Its Root Graph“. In: *Journal of the ACM (JACM)* 21.4 (Okt. 1974), S. 569–575. ISSN: 00045411.

[LH09] Jan Langer und Ulrich Heinkel. „High Level Synthesis Using Operational Properties“. In: *Forum on Specification & Design Languages (FDL).* IEEE, Sep. 2009, S. 1–6.

[LPH10] Jan Langer, Dimo Pepelyashev und Ulrich Heinkel. „Determinierung von Automaten bei der High-Level-Synthese von Operationseigenschaften“. In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV).* Hrsg. von Manfred Dietrich. Fraunhofer Verlag, 2010, S. 31–40.

[LS91] Charles E. Leiserson und James B. Saxe. „Retiming synchronous circuitry“. In: *Algorithmica* 6.1-6 (Juni 1991), S. 5–35. ISSN: 0178-4617.

[MB06] Katell Morin-Allory und Dominique Borrione. „Proven correct monitors from PSL specifications“. In: *Design, Automation, and Test in Europe (DATE).* 2006, S. 1246–1251.

[MNB07] Abhishek Mitra, Walid Najjar und Laxmi Bhuyan. „Compiling PCRE to FPGA for Accelerating SNORT IDS“. In: *Architecture for Networking and Communications Systems (ANCS)*. New York, NY, USA: ACM, 2007, S. 127–135. ISBN: 9781595939456.

[MPC88] Michael C. McFarland, Alice C. Parker und Raul Camposano. „Tutorial on high-level synthesis“. In: *Design Automation Conference (DAC)*. Los Alamitos, CA, USA: IEEE, 1988, S. 330–336.

[MS09] Grant Martin und Gary Smith. „High-Level Synthesis: Past, Present, and Future“. In: *IEEE Design & Test of Computers* 26.4 (Juli 2009), S. 18–25. ISSN: 0740-7475.

[MY60] R. McNaughton und H. Yamada. „Regular Expressions and State Graphs for Automata“. In: *IRE Transactions on Electronic Computers* EC-9.1 (März 1960), S. 39–47. ISSN: 0367-7508.

[Ngu+08] Minh Duc Nguyen, Max Thalmaier, Markus Wedler, Jörg Bormann, Dominik Stoffel und Wolfgang Kunz. „Unbounded Protocol Compliance Verification Using Interval Property Checking With Invariants“. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 27.11 (Nov. 2008), S. 2068–2082.

[OC99] Étienne K. Ogoubi und Eduard Cerny. „Synthesis of checker EFSMs from timing diagram specifications“. In: *International Symposium on Circuits and Systems (ISCAS)*. Orlando, FL, USA: IEEE, 1999, S. 13–18. ISBN: 0-7803-5471-0.

[OMB09] Yann Oddos, Katell Morin-Allory und Dominique Borrione. „SyntHorus: Highly Efficient Automatic Synthesis from PSL to HDL“. In: *Very Large Scale Integration System on Chip (VLSI-SoC)*. Florianópolis, Brazil, 2009.

[OMG10] Object Management Group. *OMG Systems Modeling Language (OMG SysML), Version 1.2*. Techn. Ber. 2010.

[OSS09] OneSpin Solutions. *User Manual: OneSpin 360 MV*. Techn. Ber. 2009.

[OSS10] *OneSpin Solutions*. 2010. URL: http://www.onespin-solutions.com (besucht am 12. 02. 2010).

[OV11] *OpenVera*. 2011. URL: http://www.open-vera.com (besucht am 02. 05. 2011).

[Par07] Terrence J. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages.* The Pragmatic Bookshelf, 2007.

[PCI95] PCI Special Interest Group. *PCI Local Bus Specification.* Techn. Ber. 1995.

[Pep09] Dimo Pepelyashev. „Konstruktion von deterministischen Automaten des Kontrollflußgraphen bei der High-Level-Synthese von Operational Properties“. Diplomarbeit. Technische Universität Chemnitz, Aug. 2009.

[Pnu77] Amir Pnueli. „The temporal logic of programs“. In: *Annual Symposium on Foundations of Computer Science (SFCS).* Providence, RI, USA: IEEE, Sep. 1977, S. 46–57.

[PP06] Nir Piterman und Amir Pnueli. „Synthesis of Reactive(1) Designs“. In: *Verification, Model Checking, and Abstract Interpretation (VMCAI).* 2006, S. 364–380.

[PR89] Amir Pnueli und Roni Rosner. „On the synthesis of a reactive module“. In: *Principles of Programming Languages (POPL).* New York, NY, USA: ACM, Jan. 1989, S. 179–190. ISBN: 0897912942.

[Rab72] Michael Oser Rabin. *Automata on Infinite Objects and Church's Problem.* Boston, MA, USA: American Mathematical Society, Jan. 1972. ISBN: 0821816632.

[Rol87] Timothy J. Rolfe. „On a fast integer square root algorithm“. In: *ACM SIGNUM Newsletter* 22.4 (1987), S. 6–11. ISSN: 0163-5778.

[Saf88] Shmuel Safra. „On The Complexity of Omega-Automata“. In: *Foundations of Computer Science (FOCS).* IEEE, 1988, S. 319–327.

[Sah+07] Hans Sahm, Joachim Knäblein, Andreas Preiß, Mathias Koppin, Jan Langer, Ingmar Seifert und Daniel Froß. „Die Notwendigkeit formaler Methoden für Kommunikations SoCs“. In: *Kooperations- und Fachworkshop Verifikation.* Hannover, Germany: edacentrum, 2007.

[Sch09] Martin Schickel. „Applications of Property-Based Synthesis in Formal Verification“. Dissertation. Technische Universität Darmstadt, 2009.

[Sch10] Christian Schrader. „FPGA Implementierung eines Partikelfilters“. Diplomarbeit. Technische Universität Chemnitz, 2010.

[She05] Mary Sheeran. „Hardware Design and Functional Programming: A Perfect Match“. In: *Journal of Universal Computer Science (JUCS)* 11.7 (2005), S. 1135–1158.

[She84] Mary Sheeran. „muFP, a Language for VLSI design“. In: *LISP and Functional Programming (LFP)*. Austin, TX, USA: ACM, 1984, S. 104–112.

[Sie05] Robert Siegmund. „Ein Verfahren zur Spezifikation von Protokollen für die Verifikation und Synthese protokollorientierter digitaler Systeme“. Dissertation. Technische Universität Chemnitz, 2005.

[SP01] Reetinder Sidhu und Viktor K. Prasanna. „Fast Regular Expression Matching using FPGAs“. In: *Field-Programmable Custom Computing Machines (FCCM)*. Apr. 2001, S. 227–238.

[Ste90] Guy L. Steele Jr. *Common LISP: The Language*. 2nd Ed. Digital Press, 1990. ISBN: 1555580416.

[TBF05] Sebastian Thrun, Wolfram Burgard und Dieter Fox. „The Particle Filter“. In: *Probabilistic Robotics*. MIT Press, 2005. Kap. 4.3, S. 96–113.

[Urd+10] Joakim Urdahl, Dominik Stoffel, Jörg Bormann, Markus Wedler und Wolfgang Kunz. „Path Predicate Abstraction by Complete Interval Property Checking“. In: *Formal Methods in Computer-Aided Design (FMCAD)*. Lugano: IEEE, 2010.

[Var01] Moshe Y. Vardi. „Branching vs. Linear Time: Final Showdown“. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2001, S. 1–22.

[WT85] Robert A. Walker und Donald E. Thomas. „A model of design representation and synthesis“. In: *Design Automation Conference (DAC)*. ACM, 1985, S. 453–459.

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Thesen

1. **(Zeitdiagramme)**
   In Spezifikationen genutzte Zeitdiagramme sind eine geeignete Beschreibungsmethode für das Verhalten von digitalen Schaltkreisen. Dies ist insbesondere dann der Fall, wenn das Verhalten in Operationen fester Zeitdauer unterteilt werden kann, sodass sich das Gesamtverhalten durch eine Aneinanderreihung der Operationen ergibt.

2. **(Entwurfsebene)**
   Die Entwurfsebene einer mit Hilfe von Operationen erstellten Schaltungsbeschreibung ist bezüglich ihres Abstraktionsgrades zwischen der Register-Transfer-Ebene und der algorithmischen Ebene anzusiedeln.

3. **(Vollständigkeit)**
   Die Anwendung des Verfahrens der vollständigen formalen Verifikation eines Satzes von Operationseigenschaften sorgt für eine höhere Güte einer aus den Eigenschaften synthetisierte Schaltung. Dies ist insbesondere dann der Fall, wenn der Erzeugung des Eigenschaftssatzes kein zu verifizierender Entwurf zugrunde liegt.

4. **(Annahme)**
   Unter der Annahme, dass ein auf Vollständigkeit geprüfter Eigenschaftssatz implementierbar und eindeutig ist, kann automatisiert ein den Eigenschaften entsprechender Schaltkreis synthetisiert werden.

5. **(Makros)**
   Die in den Operationseigenschaften für die Beschreibung des Datenflusses verwendeten Makros nutzen einen funktionalen Beschreibungstil. Deshalb existiert ein effizientes und transparentes Verfahren, um sie in eine entsprechende Hardwaredarstellung umzuwandeln.

6. **(Nichtdeterministischer Automat)**
   Das Ergebnis der Synthese ist ein nichtdeterministischer endlicher Automat (NEA), dessen Hardwarerepräsentation ebenso viele Speicherelemente benötigt, wie der Automat Zustände besitzt.

7. **(Determinierung)**
   Es existiert ein effizientes Verfahren, das den NEA in einen deterministischen endlichen Automaten (DEA) überführt, wobei dabei nur auf die Annahme aus These 4 zurückgegriffen wird. Der Verbrauch an Speicherelementen für die Darstellung des Automaten verhält sich logarithmisch zur Anzahl der Zustände im NEA.

8. **(Inkrementelle Konstruktion)**
   Die Erzeugung beider Automaten kann inkrementell geschehen, sodass nicht erreichbare Zustände keine Auswirkung auf die Kenngrößen des Verfahrens haben.

9. **(Lange Eigenschaften)**
   Mit Hilfe von geeigneten heuristischen Verfahren ist eine Optimierung der erzeugten Schaltung möglich, die es ermöglicht auch sehr lange Eigenschaften mit bis zu mehreren tausend Zeitschritten in eine effiziente Hardwaredarstellung zu überführen.

10. **(Komplexität)**
    Laufzeit und Speicherverbrauch der beschriebenen Verfahren, sowie Ressourcenverbrauch und Zeitverhalten der erzeugten Schaltungen bewegen sich für die meisten praktischen Anwendungsfälle im akzeptablen Rahmen, d.h. es tritt keine exponentielle Zunahme dieser Kenngrößen auf.

11. **(Konzeptionelle Zustände)**
    Die bei der formalen Verifikation von Operationseigenschaften notwendigen konzeptionellen Zustände, die den Übergang zwischen zwei Eigenschaften darstellen, können bei der Synthese entfallen.

12. **(Eigenschaftsprüfung)**
    Der erzeugte Schaltkreis kann mit Hilfe der für die Synthese verwendeten Eigenschaften formal verifiziert werden. Dadurch werden die Annahmen aus These 4 formal überprüft und eventuelle Fehler im Syntheseprozess ausgeschlossen.

13. **(Logiksynthese)**
    Die auf der erzeugten Schaltkreisbeschreibung durchgeführte Logiksynthese muss insbesondere die Verfahren des *Retiming* und des *Resource Sharing* beherrschen, um einen effizienten und ressourcenschonenden Schaltkreis zu ermöglichen.